AD690446

NWL Technical Report No. 2281

April 1969

A HEURISTIC PROGRAMMING SYSTEM

by

David K. Jefferson

Warfare Analysis Department

Distribution of this document is unlimited.

FOREWORD

This work was conducted in the Programming Systems Branch of the Compute. Programming Division. The project was supported by the Independent Exploratory Development Program in Computer Based Information Processing. This paper is the first of several reports to be presented concerning a computer-aided research tool called "A Heuristic Programming System". This report discusses the philosophy, structure and use of the System. Future reports will describe the implementation, application, and evaluation of the System.

The report has been submitted to the University of Michigan, Ann Arbor, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer and Communications Sciences.

RELEASED BY:

A. L. Jones

A. L. JONES

Head, Warfare Analysis Department

ABSTRACT

The Heuristic Programming System is a tool for research in many areas of artificial intelligence, particularly pattern recognition and adaptive systems. It provides the arithmetic capabilities and recursive structure of ALGOL plus flexible and efficient facilities for representing and manipulating complex hierarchically structured objects. Objects may be created, modified, destroyed, or described by other, descriptive, objects. A search operation can retrieve objects or collections of objects which are specified by arbitrarily complex descriptions. Another search operation can not only retrieve objects, but can construct them according to the specifications of previously created descriptive objects; this greatly facilitates the implementation of self-improving pattern recognition schemes, which are basic to advanced work in artificial intelligence.

The report contains a discussion of the programming facilities required for artificial intelligence, an informal introduction to the System, a formal programmer's manual with numerous examples, a sample program which plays the game of Go-Moku, and a discussion of a proposed implementation.

TABLE OF CONTENTS

	Page
INTRODUCTION	1
CHAPTER I. PROBLEM SOLVING	3
Representation of objects; control of program flow; improvement; overview of the System; facilities for representation; control mechanisms; facility for improvement	
CHAPTER II. THE HEURISTIC PROGRAMMING SYSTEM: I	18
The Class Language; representation; control:I; the Problem Solving Executive; control:II; improvement	
CHAPTER III. A SAMPLE PROGRAM	35
Description of Go-Moku; description of the Go-Moku program; the Go-Moku program	
CHAPTER IV. THE HEURISTIC PROGRAMMING SYSTEM: II	60
Data structures and Data Management; more about the Problem Solving Executive; the Communicator; system procedures	
CHAPTER V. IMPLEMENTATION	71
CHAPTER VI. CONCLUSION	75
BIBLIOGRAPHY	81
APPENDIX	
A. Distribution	

LIST OF TABLES

Table		Page
1.	Standard Properties	19
2.	System Procedures	69

LIST OF ILLUSTRATIONS

Figure		Page
1.	Various Go-Moku Positions	36
2.	The Best Laid Schemes	40
3.	Main Flow and Legend	64
4.	Detailed Flow	65
5.	COMPSTRUCT PROCEDURE	66
6.	More COMPSTRUCT	67

INTRODUCTION

The Heuristic Programming System is an attempt to construct a high-level language within which a problem solver may, as naturally as possible, express his ideas to the computer. The purpose is not to converse with the computer in a natural language, but rather in a problem-solving language; just as the mathematician uses a highly specialized and formalized language in expressing his mathematical ideas, so should the problem solver have his own specialized language. Unfortunately, the problem solver is concerned not only with the solution, but with efficiency as well; thus, he must express a great many different kinds of information to the computer. This fact considerably complicates the language, which would be complicated enough anyway. The present attempt - the Heuristic Programming System - is still rather far from the standards of conciseness and clarity set by mathematical notation, but it is, hopefully, a considerable advance in terms of what can be expressed within the language.

The presentation of the System is divided into six Chapters. Chapter I begins with an introductory discussion of the general requirements which a problem solving system must meet. These include flexible and efficient representations of objects and descriptions of objects, a flexible means of searching for desired objects, and a means of creating objects and descriptions of objects. Chapter I also discusses the types of problems which may be or should be solved with the System; briefly, the System is most appropriate to situations in which objects with extremely complex hierarchical descriptions must be created, destroyed, discovered, or described. Chapter I concludes with an introduction to the System. Chapter II consists of a detailed presentation of the programming language. Chapter III consists of a sample program, which plays the game of Go-Moku. Chapter IV contains a discussion of the "background" components of the System - the Problem Solving Executive, data management, communication with the user, and system procedures. Chapter IV also continues the discussion begun in Chapter I of the manner in which objects are described by the system. Chapter V is a discussion of a proposed experimental implementation of the System. Chapter VI contains a few concluding remarks.

Chapter I can be read rather quickly to grasp the general philosophy of the System. The descriptive material in Chapter II should be read carefully, with occasional reference to the sample program in Chapter III; the detailed syntax in Chapter II is intended as reference material, so it need not be studied closely during the initial investigation of the sytem. Points which remain unclear after Chapters II and III may be clarified in Chapter IV, particularly within the area of data management. Chapter V need not be read at all, unless the reader is concerned with the feasibility of the System or with actually implementing it himself.

CHAPTER I
PROBLEM SOLVING

Discussion of the Heuristic Programming System will be organized around three somewhat overlapping topics: representation, control, and improvement. Briefly, "representation" refers not only to what *can* be stored within an information processing system, but the *manner* in which it is stored and accessed. Thus, representation influences both the theoretical and practical limitations of a system. "Control" refers to the manner in which operations and operands are chosen. "Improvement" refers to improvement of the choice mechanism. These topics and their interrelations will be discussed both in terms of this Heuristic Programming System and in terms of previous heuristic programs.

Representation of Objects

In the following discussion, the term "object" will be used to denote any data which might be pertinent to the solution of a problem. The term includes those data which are used to describe the problem (e.g., the location of a piece in a game-playing problem, or the end-point coordinates of a line in a pattern recognition problem), those data which are derived from data defined by the problem and which may be relevant to its solution (e.g., a subgoal which has been recognized, but whose reliability as an indicator of progress is questionable), and those data which represent descriptions of other objects (e.g., a description of a subgoal).

Flexibility

The most important step toward the solution of a complex problem is easily and efficiently representing all that is known or guessed about it. If a problem is simple, almost any solution method may suffice; added data may even hamper the solution. However, if the problem is complex it probably will not be solved at all unless all available data are utilized. Minsky (1961) makes the point that simple problems are qualitatively quite different from complex problems: simple problems can be solved by trying out all possible actions, while it may be impossible to try out even the most promising actions in solving a complex problem (indeed, it may be impossible even to find all the promising actions). Amarel (1968) discusses

various ways of representing the missionary and cannibals problem, and how the representation effects the efficiency of the solution. The point is not that there are ways to find a good representation, but that, if one is known, it should be used: many of the present difficulties with heuristic programs arise because of inappropriate and inflexible representation of the problem, while the notable successes have occurred because (by chance or design) the available structure fit the "natural" structure of the problem. Ernst and Newell (1967) discuss these difficulties in describing the drastic changes necessary to enable GPS (Newell, Shaw, Simon, 1960a) to solve a wider class of problems. Basically, the original GPS was designed around a certain type of problem; the heuristic power which it brought to bear on theorem proving, while sufficient to solve other types of problems, could not be used because of severe problems of representation. (Although the new version of GPS is greatly superior to the old version in terms of the goals, operators, and differences which *can* be represented within GPS, it still suffers from a lack of flexibility in terms of the *manner* in which these objects are represented. Hence, because of time and memory limitations, GPS cannot in practice solve problems which are solvable in principle.)

The concept formation problem described by Amarel (1961) is an even more striking example of the power to be gained by proper representation of a problem. Amarel's scheme is fairly complex - it actually involves two levels of language - but he gains the tremendous advantage of being able to represent program schemata, which enables him to make "subroutines" out of complex sequences of operators. The important point is not that Amarel's representation is general - it is not - but that it, to a great extent, determines the power of his system.

The system devised by Hormann (described in its latest form in 1965; additional details of development are found in 1962 and 1964) involves not only problem solving of a fairly sophisticated degree but planning and induction as well. The adequacy of the representation becomes vitally important in her Induction Mechanism, which is a general pattern recognizer and conjecture generator. Clearly, if the patterns which are recognized have nothing to do with the solution of the

problem, then the Induction Mechanism must fail. The regularities of the problem must, then, be representable to the various problem solving and learning mechanisms. This is accomplished by special routines which abstract and characterize a particular problem. The result is similar to the differencing scheme of GPS, with the important addition of a subroutine feature somewhat like that of Amarel's scheme.

A final example of the power of a well-chosen representation is the game-playing program of Koffman (1967, 1968). Koffman represents a position in a simple class of games (which includes Qubic, Hex, Bridge-it, and a simplified version of Go-Moku) in terms of lines of pieces and intersections of lines. This enables the program to easily describe a subgoal (a board configuration from which a win can be forced) in terms of lines with specified numbers of empty places and intersections with each other. Representation of the various symmetric or translational images of such a subgoal is automatic, because the representation is unchanged by such transformations.

The Heuristic Programming System is designed to enable the user to easily and efficiently represent hierarchical, or tree-structured, objects - i.e., objects which consist of subobjects, which consist of subsubobjects, and so on. Objects of this sort are obviously essential to pattern recognition, which is an important area in itself as well as a major subproblem in the solution of any complex problem. Hierarchical objects are of fundamental value for two additional reasons: such objects form a basis for subdividing the problem into simpler problems, which are solved more or less separately, and such objects are essential to the human programmer, so that he can easily write, debug, and alter his program. For problems in which the majority of the objects are hierarchical, the System offers the user tremendous ease and flexibility of representation: an object may be described (by another object), created (by explicit specification in the program, or by a reference to a descriptive object), altered, or destroyed. An object may contain an arbitrary collection of objects, and may possess an arbitrary collection of numeric or Boolean properties.

Efficiency

The question of *how* to represent a collection of objects centers about the

necessity to compromise among the three conflicting requirements of representation: ease of programming, efficiency in terms of storage, and efficiency in terms of time. Ease of programming requires, primarily, that the representation should be similar to the mental representation of the problem solver, and that it should be easily alterable (especially with respect to adding detail as the user learns more and more about the problem or problem area). Storage efficiency requires that objects which must be present all the time should be represented in the most compact form possible and that objects which are needed only temporarily exist only while they are needed. Efficiency in time requires that objects should be quickly created, retrieved, modified, and destroyed, and that the effects of these operations should be easily propagated to other, dependent, objects. Note that in any problem solving program of reasonable sophistication, especially a program which involves learning or improvement, it is necessary to represent many objects (such as subgoals) implicitly - i.e., by rules for constructing these objects from classes of other objects. The retrieval of such an object may involve a great amount of searching for components which are in complex relationships to each other; once it has been retrieved, it may either be made explicit so that it can be rapidly retrieved for future use, or it may remain implicit, so that it will require no storage.

Some of the decisions necessary in choosing a proper representation have been made in the design of the Heuristic Programming System; others are left open to the user. The decisions which have been made will now be outlined; the purpose is not so much to justify them as to implicitly define the class of problems for which the System is appropriate. No claims will be made that the System is a good symbol manipulator or list processor. It is neither; the philosophy of the system is that symbols as such do not even exist, and that lists are useful primarily to the system and, therefore, the user is not encouraged to manipulate them directly.

One of the major decisions made in the internal representation of objects in the System is that all objects which are composed of other objects or which are contained in other objects must have explicit pointers to those objects. This is to facilitate searches for objects and modifications to objects: it is frequently necessary

to look for an object contained in or containing another object, either so that some decision can be made (for example, when trying to find an instance of a complex subgoal) or so that modifications to one object may properly propagate to dependent objects (for example, when a move is made in a game, many complex objects must be altered or destroyed). The necessity for these pointers does, however, create a problem of storage efficiency: twice as many pointers must be used for objects as in, for example, LISP 1.5 (McCarthy *et al.*, 1965), since for every pointer *to* a contained object there must be a corresponding pointer *from* the contained object. To somewhat compensate for this, the System allows the user to represent objects as blocks of contiguous storage, rather than as lists; this saves time as well as storage. An object can, of course, be represented as a block only if it has some properties which are assigned at its creation, and whose storage requirements cannot increase thereafter. For example, in a pattern recognition problem, a polygon might be represented as a contiguous block composed of the coordinates of the center of gravity, the number of sides, the color, and a pointer to a list of objects within which it is contained (the list might be of any length, since it might be possible to recognize any number of objects containing the polygon)

The description above applies as well to graphic data structures as to problem solving structures (cf. Roberts, 1965, Ross 1967a, 1967b, 1968, and Sutherland, 1963). This is not surprising, since the aim in either case is the efficient modeling of complex and changing situations. The differences between graphic manipulations and problem solving lie in the control and improvement facilities: this is due to the fact that the graphic facilities are algorithmic (hence a process will terminate by itself), while the problem solving facilities are heuristic (and hence a process will need supervision to make sure that it will not continue endlessly).

Control of Program Flow

As stated earlier, "control" refers to the manner in which operations and operands are chosen. Two processes are important: "understanding" the current situation, and deciding how to achieve a "better" one. This section will attempt to

present the major issues involved in these two processes, how they are implemented in the System, and their relation to representation.

Allocation of effort

Understanding a complex situation rather clearly implies the need for describing and recognizing complex objects. In simple problems, one may be able to explicitly represent each relevant object, so recognition is simple. This is not possible for complex problems; it is necessary to search for objects which are represented implicitly (i.e., by descriptions of how they are constructed), since storage limitations prohibit explicit representation of all but a small fraction of the possibly relevant objects. Furthermore, it might be impossible to generate all such objects within a reasonable length of time even if storage were unlimited. Searching for objects which are represented implicitly may take an indefinitely large amount of time; hence, it should be possible to specify how much effort is to be allocated to each search.

Attempting to achieve a better situation involves deciding what to do next, doing it, and then, as above, evaluating the result to see if progress has been made. In problems where actions are not irrevocable it may be as important to decide how much of the available resources of time and/or storage are to be devoted to each alternative as to decide which is to be attempted first. Deciding what alternatives are to be considered may itself be a non-trivial task, requiring a special allocation of resources, as in the generatemove procedure of the Go-Moku program in Chapter III. There are various well-known strategies for choosing among alternatives and allocating resources: for example, trying to reduce the most important difference between the present situation and a goal (Newell, Shaw, and Simon, 1960), trying to attain locally maximal results, concentrating effort on the most sensitive regions, and equalization of effort among alternatives (Amarel, 1962).

From this discussion it is clear that allocation of effort is a major issue in solving complex problems. The Heuristic Programming System provides two methods of searching for objects: one is a straightforward search down a list to see if an object exists, the other is a search for an implicitly represented object. The latter

search is limited to a program-specified amount of time, which greatly simplifies the coding of complex resource-allocation schemes. This search mechanism may attempt to combine objects to form an object (or class of objects) which is specified either by the programmer or by a previously created description (cf. the discussion of improvement); this facility, of being able to describe how an object is to be constructed, also greatly simplifies the coding of a complex program.

Search strategies

In solving complex problems, it is frequently necessary to be able to try many different alternative actions from a given situation: therefore, it is necessary to decide first what situation to start from, and second what action to attempt. Three common strategies are breadth-first (i.e., simultaneously try all possible actions from all current situations, as in the Logic Theory Machine, Newell, Shaw, Simon, 1967), depth-first (i.e., pursue one series of actions to a terminal point, then go back to the nearest untried alternative, try that, etc., as in GPS, Newell, Shaw, Simon, 1960), or maximum payoff (i.e., choose the situation and action which are most promising, as in Slagle and Bursky, 1968). The first strategy is rather uninteresting: storage and time requirements are too high. The two other strategies both require the preservation of intermediate situations: that is because, in order to attempt an action from a situation, the objects which make up that situation must be present in storage and must be distinguished from the objects of other situations. Hence, it is necessary either to preserve all objects simultaneously (difficult even for some of the simple problems of GPS, as noted in Ernst and Newell, 1967; impossible for complex problems involving thousands of highly interacting objects) or to reconstruct the desired situation (either by reversing actions or by repeating actions from the initial situation or from some intermediate situation which has been preserved). Note that the maximum payoff strategy will in general explore fewer alternatives than the depth-first strategy, but it will consume more time in reconstructing situations. A reasonable compromise between the two would be to weight each alternative according to how cheap it would be to attempt it; this would lead to a somewhat "stubborn" maximum payoff strategy, which might be of great interest

psychologically. Such a strategy would be fairly easy to implement using the Heuristic Programming System.

The search technique used in the Go-Moku program is alpha-beta minimax, a fairly common strategy in game playing. The minimax is used to evaluate a given position by evaluating the positions which could result from it: the value of a position is defined as either the value of some evaluation function if the position is "terminal" (e.g., if a maximum search depth has been reached, or if the position is relatively stable), or as the "best" choice among the values of the successor positions. The "best" choice clearly depends upon whose move it is. Not quite so clearly, it is possible to avoid evaluating certain positions if it can be shown that they will never be reached in rational play. For example, assume that the program has found that, from a given position, it can attain a position of value at least alpha, despite best play by the opponent. Assume now that the program is evaluating a possible move, M, and that one of the opponent's replies, R, proves to have value less than alpha. Then it is not necessary to evaluate any more replies: if move M is made, then the opponent can make a reply at least as powerful as R, so the result will have value less than alpha. Search can therefore be terminated by an "alpha cutoff" (a more lengthy discussion is given by Slagle and Bursky, 1968). A "beta cutoff" is obtained if the positions of the program and opponent are reversed in the above example. If the possible moves are ordered so that the most promising are evaluated first, then a great deal of time will be saved by such alpha-beta cutoffs; hence, a very important part of the Go-Moku program is the generatemove procedure; which chooses feasible moves and orders them according to estimated value. The program can be expected to make more cutoffs with practice, since the ordering of moves will become more accurate.

This sort of strategy was chosen for the Go-Moku program because it is fairly easy to implement, reasonably efficient (at least, after experience has generated useful subgoals), and requires much less construction and reconstruction of objects than the maximum payoff strategy. Since the strategy is rather simple, it was possible to use some specialized but efficient techniques to minimize the time

required to reconstruct objects; see the discussion in Chapter III.

Improvement

As stated earlier, "improvement" refers to the improvement of the mechanism by which operations and operands are chosen. The fact that this definition is equivalent to common definitions of learning and adaptation follows from the previous discussion of control. In the Heuristic Programming System, objects may be constructed according to other, descriptive, objects, and may then be used to evaluate a situation and decide upon the next action. Facilities for creating, modifying, and destroying descriptive objects are provided by the System: this section will present some of the issues involved in using these facilities to implement a self-improving problem solving program.

A basic technique for finding a subprogram which successfully solves a problem is to search in neighborhoods of partially successful subprograms: this implies that partial success must be recognizable, and that neighborhoods must be definable such that the degree of success does not vary too greatly within a neighborhood. Improvement then consists of fairly straightforward hill-climbing (Minsky, 1963). However, it is frequently impossible to recognize partial success or to define neighborhoods, so the original problem area must be transformed into a new area, or model, in which these things can be done. In that case, trials in the old area indicate the degree of success in the model; neighborhoods in the model indicate where trials should be conducted in the old area. For example, Samuel (1959) transforms the problem of choosing a good checker move into the problem of finding an optimal n-tuple of weights of features of positions, Holland (1962) transforms the problem of constructing problem solving programs in an iterative circuit computer into a scheme for the differential selection of "supervisory programs", and Newell, Shaw, and Simon (1960b) transform classes of symbol-manipulation problems into the construction of sets of differences and the determination of which operators reduce which differences.

Obviously, the utility of this technique depends critically upon the degree to which success in the old area corresponds to success in the new. Thus, Samuel's

program can improve its performance only to a certain degree. Further progress requires the recognition of new features and dependencies between features; in other words, a more complex model. The improvement facilities of the Heuristic Programming System might be used in either of two ways: to produce descriptions of sets of features, so that dependencies could be detected, and to produce descriptions of board configurations, in order to recognize abnormal situations in which feature evaluation is unreliable.

An important consideration in solving practical problems is being able to utilize specialized knowledge to reduce search time. This is difficult in Holland's scheme because such knowledge must be introduced very carefully in order not to disrupt the general improvement scheme; programs which enjoy great initial advantages must not cause the premature rejection of programs whose potential is great but not easily achieved.

A related difficulty arises in the Newell, Shaw, and Simon scheme, which is limited by the fact that the operators which recognize differences are completely distinct from the operators which manipulate objects (and which therefore generate and remove differences). Hence, it may be very difficult to introduce differences whose values are known to be great. It may also be quite difficult to utilize specialized operators, because the program may not be able to learn when to apply them.

The Heuristic Programming System provides facilities for describing and recognizing situations in which specialized knowledge might be used, and it also provides facilities for improving such descriptions. Hence, as a program's generalized knowledge grew, it could continually change its methods of integrating this knowledge with the specialized knowledge.

Finally, the System provides a framework within which one could construct a hierarchical control scheme with a growing number of levels (unlike the limited number of levels in the above-mentioned schemes); this is rather critical to really advanced self-improving programs, because these must be able to cope with the "aha! phenomenon" - the sudden integration of a number of facts on one level into

a new concept on a higher level. The System provides a facility for constructing a description of any object, including another description, so "all" the user need do is to program criteria for deciding when to make new descriptions, and when to search for instances of the newly described objects.

Overview of the System

The Heuristic Programming System consists of the following major sections:

1. The Class Language (CL),
2. The Problem Solving Executive (PSE),
3. The Data Management Routine,
4. The Translator, and
5. The Communicator.

CL is the programming language in which the user describes the structure of a problem area: it is a considerably extended ALGOL which enables the user to create, manipulate, and destroy sets and classes of sets. The external form of the language has been greatly influenced by LEAP, the Language for the Expression of Associative Procedures (Rovner and Feldman, 1967), which is an ALGOL based language with facilities for manipulating sets and associative data. However, both the purpose and internal structure of CL differ greatly from LEAP.

Facilities for Representation

Set, class, goal, description, and property

The basic non-ALGOL structure in CL is the *set*. Sets with common properties may be combined together into a *class*; the properties may have different values for different sets in the class. Class members may be created or destroyed during a computation. A class may be declared to be a *goal class* or a *descriptive class*: the PSE may construct members of a goal class from other sets, using members of a descriptive class as descriptions of the goal class.

A set may be accessed in various ways: by name, by reference from a set contained within it, or by reference from a set within which it is contained. If a set is a member of a class, then its name is the name of the class followed by an index (i.e., a subscript). The user may insert a series of integer-valued expressions

between the name and index, indicating subclasses.

Creation and destruction

Any set which is declared within a block is created upon entering the block, just as a local variable in ALGOL; sets are created empty. The members of a class are created when entering a block if the class has a fixed cardinality (i.e., number of members); otherwise members are created by the *create* procedure. Sets are destroyed when exiting from the block in which they are declared: class members may also be destroyed by the *destroy* procedure. The destruction of a set means that all properties of the set become undefined. Further, *class members which contain a destroyed member are also destroyed;* all *references* from sets *contained* in the destroyed member are deleted. Numeric or Boolean properties may be defined in any of three ways: a constant initial value may be assigned during the compilation of the block in which a set is defined, or an expression may be assigned at compilation time and then evaluated at the creation of the set, or the value may be assigned within the block, overriding any previously assigned value.

The Data Management Routine

The Data Management Routine has two primary functions: to create data structures within the free storage area, and to return destroyed data structures to free storage. For each class, there is a block of contiguous storage, called the class header, which contains information relevant to the class as a whole: the class name, the names of the properties, initial values of properties (or pointers to expressions for determining the initial values), and the method of ordering the members. The class has either a fixed number of members (in which case the header contains a block of class members), or a variable number of members (in which case the class header contains, *for each subclass*, the number of members, the highest index, the index of the most recently accessed member, and pointers to the most recently accessed member, the member with lowest index and the member with highest index). Each class member consists of a block of contiguous storage which contains the values of the member's properties, the member's index and a pointer to the member with next higher index, either a pointer to a list of contained elements or

a block of pointers to each contained element, and either a pointer to a list of containing elements or a block of pointers to each containing element. Sets are comparatively simple: each consists of a set header containing the set name, the number of members, the method of ordering, the property values, and either blocks of pointers to the contained and containing elements, or pointers to lists which contain pointers to the contained and containing elements.

Thus the data are seen to fall into two categories: blocks of contiguous storage of various sizes, and simple lists linking the blocks together; both types of data are sufficiently simple to be manageable by a relatively unsophisticated routine which, on demand, obtains storage or returns blocks or lists to free storage. A discussion of a more general system, which influenced this design, is given by Ross (1967a, 1967b). The primary advantage of this sort of data structure over, for example, a pure list structure, is that the tighter grouping of data (by means of contiguous storage or many pointers) greatly reduces the amount of time to search from one item of information to another. In some cases, contiguous storage saves space; the "line" in the Go-Moku program, for example, always contains exactly five "points", so that the structure can be represented by a block of five pointers, rather than a list with five elements.

A useful feature of the Heuristic Programming System is that it is easy for the user to write a specialized storage reclamation routine: thus, he can delete less valuable sets, or sets of certain types, or even, by means of the destroy statement, all sets whose existence depends upon a given set.

Control Mechanisms

There are three basic types of control mechanisms in the Heuristic Programming System: basic ALGOL (which will not be discussed further), the CL for (a search mechanism), and the PCE (a descriptive and constructive mechanism).

The CL for

The CL for is a sophisticated search facility. The user writes a *template* which is basically the specification of the class name of an object, its structure (or various possible structures), and its properties. The class name, and any of the class names

appearing in the structural description, may be followed by a "defined" index or by an "undefined" variable. The defined index refers to a specific class member, while the undefined variable refers to a member which is to be found by the search. When the desired member is found, its index is assigned to the variable; the member then can be manipulated by reference to its class name and index.

Thus, the basic orientation of the Class Language is toward explicit *specification* and *search*; the user is spared the task of *choosing* and *identifying* objects. This tends to make CL code reflect the user's mental representation of objects, which facilitates coding and debugging.

The CL for has been influenced by both LEAP (Rovner and Feldman, 1967) and SNOBOL (Farber, Griswold, and Polonsky,1964, and Forte,1967).

The PSE

A CL for is used to search for objects which exist, i.e., which have been constructed previously; a PSE statement may not only *search* for objects, but it may also *combine* objects, according to templates, in order to *construct* members of goal classes (after a goal object has been constructed, it can be manipulated by ordinary CL code). The PSE is, therefore, goal-directed, like the productions of the COGENT programming system (Reynolds, 1965) and other syntax-directed translating systems (Feldman and Gries, 1968). Unlike other systems, however, the PSE can construct more than one object, or it may not be able to construct any at all. Each invocation of the PSE must, therefore, specify the maximum amount of time to be spent, and whether one or many objects are desired.

Facility for Improvement

The most important control features of the Heuristic Programming System are its methods for searching for specified objects; therefore, improvement consists of creating, destroying, and altering descriptions of objects. Members of descriptive classes may be created and destroyed just as members of other types of classes, but the facilities for alteration are unique. Basically, new descriptions are produced in the following manner: a goal class member is created, then a descriptive member is created which contains the goal class member, the descriptive member may be *edited*

and then the descriptive member is *abstracted*.

Editing is a process by which a description is changed without changing the thing described; each set which is to be changed is copied, then the copy is changed. All references to the original set within the description are changed into references to the copy. Abstracting is a process of copying all sets, as above, and then changing the set names into undefined names. Thus, the result is just a template.

Other Sections of the System

The Translator translates the CL program into a legitimate ALGOL program. Procedure calls are inserted at appropriate places in the code to call the PSE and Data Management. More details on the operation of the Translator are given in Chapter V, in the discussion of implementation.

The Communicator is both the outermost block of the user's program, which contains declarations for the system variables and the code for system procedures, and a collection of procedures for communication with the user. The user may request the creation, destruction, or display of any object within his program. Objects may be described explicitly by, for example, name and index, or implicitly by means of a template, just as in ordinary CL code. The correspondence between the internal and external names of classes and sets is provided by the symbol table produced during translation, and by the names which appear in the class and set headers.

CHAPTER II

THE HEURISTIC PROGRAMMING SYSTEM:I

The Class Language

The detailed presentation of CL will assume a moderate knowledge of ALGOL and the metalinguistic formulae used to describe ALGOL. In particular, the revised ALGOL report (Naur *et al.*, 1963) defines formally various metalinguistic variables which are used here; these are, for the most part, self-explanatory. Both the metalinguistic variables used to describe ALGOL and those used to describe the additional constructs of CL are indicated in this Chapter by the brackets "(" and ")"; generally there will be no difficulty in distinguishing ALGOL from the new constructs.

Representation

Declaring sets and classes

A name may be declared to be a set name by the declarator **set** or a class name by the declarators **class**, **goal class**, or **descriptive class**. Sets are normally ordered, but may be declared to be **unordered**, **rankhi** (high member first), or **ranklo**. Ranking may be on the basis of any property of the members, or, in the absence of any contrary specification, by value, which is a standard property (see Table I). The members of a class may be declared to be **unordered**, **rankhi** or **ranklo** in the class declaration (the class, itself, is always ordered). The cardinality of a class may be established at the time that the class is declared, as in the following example:

class class [100]

This class is not empty initially; it consists of as many members as are declared, although these members need not have any defined properties.

A class may be subdivided into subclasses, subsubclasses, and so on, to any desired extent. This is indicated in the declaration as, e.g.,

class subclasses [10,5];

Here, there are 10 subclasses, each with five members.

TABLE 1

STANDARD PROPERTIES

Name	Possible values	Explanation
structure	a list of sets	The structure of a set is a list of the elements contained in it; this is *not* the same as the set itself: if A contains B, and C contains the structure of A, then C contains B but not A.
cover	minval-maxval;	The sets containing a given set.
value	minval is default	The Communicator sets minval to 0. and maxval to 1.; these may be altered in the program. The value property is standard only for goal or descriptive classes. The PSE uses values to decide which goal to attempt.
cost	any real number	This is either the estimated cost of constructing a member of a goal class, or the actual cost of having constructed a goal object. This is used by the PSE.

If a cardinality is not declared, then a class may have any number of members. This flexibility may be purchased at a cost in processing time, since an array structure is used for a class of fixed cardinality, while a list structure is necessary in the general case. A specific element of a class of fixed cardinality is accessed by "pointing" to it; otherwise the list of members must be searched to find it. The search time is, however, quite small in many cases, because a record is kept of the location (on the list) of the most recently accessed member of each class; this greatly facilitates operations which sequence through the class or which repeatedly refer to the same member.

Note that if a class has a declared cardinality, then any member may be referred to (e.g., may have values assigned to its properties) without explicitly creating it. If a class has ariable cardinality, on the other hand, each member must be explicitly created by a create statement before referring to it. Thus. the two types of classes are quite different.

A class may be declared to have subclasses even though it has no fixed cardinality. Thus,

class manymember [5,];

declares a class which has 5 subclasses, each with any number of members.

Declaring properties

The standard properties structure and cover are automatically declared whenever a set or class is declared (see Table 1). The number of elements in either the structure or the cover may be declared. For example,

class triangle (integer structure:=3, cover:=5);

declares a class of triangles, each member of which contains at most three elements and is contained in at most five elements. Such a declaration means that the pointers to the elements of the structure and cover may be placed in a contiguous block in each class member, rather than in lists. The value and cost properties are automatically declared for goal and descriptive classes.

Nonstandard properties must be declared explicitly, and may be assigned initial values. Thus

class box (real length, width:=2., height:= p + q);

specifies that each member of the "box" class has a length, a width (with value 2. when the member is created), and a height (with value p + q when the member is created, for the then current values of p and q).

Syntax of declarations

The syntax of declarations will now be given. This defines not only the special CL constructs, but their relation to ALGOL as well.

⟨ declaration ⟩ ::= ⟨ type declaration ⟩ |⟨array declaration ⟩ |
⟨ switch declaration⟩ |⟨procedure declaration ⟩ | ⟨set
declaration ⟩ | ⟨class declaration ⟩ | ⟨ goal declaration ⟩ |
⟨ descriptive declaration ⟩ |⟨standard test⟩

Syntax of set declarations

⟨set name⟩ ::= ⟨ identifier ⟩
⟨property name ⟩ ::= ⟨identifier ⟩
⟨short property assignment⟩ ::= ⟨ property name ⟩ := ⟨arithmetic
expression⟩ |⟨property name⟩:= ⟨Boolean expression ⟩
⟨ property item⟩ ::= ⟨ property name⟩ |⟨short property assignment ⟩
⟨ property list⟩ ::=⟨property item⟩ |⟨property item⟩ , ⟨property list⟩
⟨ typed property list⟩ ::= ⟨type⟩ ⟨property list⟩ |⟨type⟩ ⟨property
list⟩, ⟨typed property list⟩
⟨ property declaration⟩ ::=⟨empty⟩ |(⟨ typed property list⟩)
⟨ set declaration item⟩ ::= ⟨ set name⟩ ⟨ property declaration ⟩
⟨ set declaration list ⟩ ::= ⟨ set declaration item⟩ | ⟨set
declaration item⟩,⟨ set declaration list ⟩
⟨rank ⟩ ::= **rankhi**|**ranklo**
⟨ modifier ⟩::= **unordered** | ⟨empty⟩ | ⟨ rank ⟩ |⟨rank ⟩
by ⟨variable ⟩
⟨set declaration⟩ ::= ⟨modifier⟩ **set** ⟨ set declaration list ⟩

Syntax of class declarations

⟨ basic class name ⟩ ::= ⟨ identifier ⟩

⟨ subclass and cardinality list ⟩ ::= ⟨ arithmetic expression ⟩ |
⟨ arithmetic expression ⟩ , ⟨subclass and cardinality list ⟩

⟨ subclass and cardinality declaration ⟩ ::= ⟨ empty⟩ | [⟨subclass and cardinality list ⟩] ||⟨ subclass and cardinality list ⟩ ,]

⟨ class declaration item ⟩ ::=⟨ basic class name ⟩ ⟨ subclass and cardinality declaration ⟩ ⟨ property declaration ⟩

⟨ class declaration list ⟩ ::= ⟨ class declaration item ⟩ |⟨ class declaration item ⟩ , ⟨class declaration list⟩

⟨ class declaration ⟩ ::= ⟨ modifier⟩ **class** ⟨class declaration list ⟩

Syntax of goal declarations

⟨ goal class name ⟩ ::= ⟨ identifier⟩

⟨ goal declaration item ⟩ ::= ⟨ goal class name ⟩ ⟨subclass and cardinality declaration ⟩ ⟨property declaration ⟩

⟨ goal declaration list ⟩ ::= ⟨ goal declaration item ⟩ |⟨ goal declaration item ⟩ , ⟨goal declaration list⟩

⟨ goal declaration⟩ ::= ⟨ modifier⟩ **goal class**⟨ goal declaration list ⟩

Syntax of descriptive declarations

⟨ descriptor name⟩ ::= ⟨ identifier⟩

⟨ descriptor item ⟩ ::= ⟨descriptor name⟩ ⟨ subclass and cardinality declaration ⟩ ⟨ property declaration⟩

⟨ descriptor list ⟩ ::= ⟨ descriptor item⟩ |⟨ descriptor item ⟩ , ⟨ descriptor list⟩

⟨ descriptive declaration ⟩ ::= ⟨ modifier⟩ **descriptive class** ⟨ descriptor list⟩

⟨ class name⟩ ::= ⟨ basic class name⟩ | ⟨ goal class name ⟩ |⟨ descriptor name ⟩

Syntax of procedure declarations

⟨procedure declaration⟩ ::= **procedure** ⟨ procedure heading ⟩ ⟨ procedure body⟩ | ⟨ type⟩ **procedure** ⟨procedure heading ⟩ ⟨ procedure body ⟩ | **set procedure** ⟨procedure heading ⟩ ⟨ procedure body ⟩

Syntax of standard tests

⟨ standard test ⟩::= **standard test** ⟨ property name ⟩ | ⟨ standard test ⟩, ⟨property name ⟩

This is a simple way to avoid writing the same condition in many ⟨ template ⟩'s. A ⟨ template⟩ contains lists of class members and Boolean conditions on those members (a more complete discussion is given later). Each ⟨ property name ⟩ (which must be a Boolean variable) in the ⟨ standard test ⟩ is added as an additional condition to each member with that property, provided that no condition involving the ⟨ property name ⟩ is already present.

Referring to members of sets and classes

Members of a set or class are referred to by index and subclass indicators as, e.g.,

setname.I

classname.J

subclasses.I.J

where I and J (here and in subsequent examples) may be arbitrary ⟨ primary⟩'s and setname is any set (in particular, it could be classname.J).

Members of classes may be referred to by subclass indicators and indexes which are either "defined" or "undefined"; a period before a ⟨ primary ⟩ indicates that it is defined, as in the .I of

classname.I

while a slash before a ⟨ simple variable ⟩ indicates that it is undefined, as in the /K of

classname/K

The defined values are used to indicate a specific class member. The undefined variables are used to indicate that an index or subclass indicator is not known, as,

for example, when searching for some member with specific properties or when creating a new class member. When the desired member is found or created, its index and subclass indicators are assigned to the previously undefined variables. Similarly, a set name followed by a slash indicates an undefined set, to which structure is to be assigned; a set name not followed by a slash represents a previously defined set of elements. An asterisk followed by a period and an ⟨ unsigned integer ⟩ represents a variable to which structure has been assigned during a searching operation. This will be discussed in more detail in the section on ⟨ template ⟩'s.

Operations on sets

Sets are combined by means of the binary operators +, ×, and -, which are interpreted as concatenation (with subsequent deletion of duplicated elements, if the result is assigned to an unordered set), intersection, and subtraction. The precedence order is × first and - last, as indicated in the syntax. Association is from the left in a sequence of identical operators, or may be indicated by parentheses. Note that brackets may be used to construct sets from lists of arbitrary expressions; hence, numeric or Boolean quantities may be put into sets. Brackets are removed by the structure function so that, e.g.,

F:=[structure([A, B, C, D]),[E]];

is equivalent to

F:=[A,B,C,D,[E]];

Syntax of sets, expressions, and assignments

⟨ expression ⟩ ::= ⟨ arithmetic expression ⟩ | ⟨ Boolean expression ⟩ |
⟨ designational expression ⟩ | ⟨ structural expression ⟩

⟨ assignment statement ⟩ ::= ⟨ left part list ⟩ := ⟨ arithmetic expression ⟩ |
⟨ left part list ⟩ := ⟨ Boolean expression ⟩ | ⟨ structural assignment ⟩

⟨ index ⟩ ::= ⟨ primary ⟩

⟨ simple set ⟩ ::= ⟨ set name ⟩ | ⟨ class name ⟩ | (⟨ structural expression ⟩)
| ⟨ function designator ⟩ | *. ⟨ unsigned integer ⟩ | ⟨ simple set ⟩
⟨ index ⟩

⟨set⟩::= ⟨simple set⟩ | [⟨set list⟩] | empty

⟨set list⟩ ::= ⟨expression⟩ | ⟨expression⟩, ⟨set list⟩

⟨set factor⟩ ::= ⟨set⟩ | ⟨set⟩ X ⟨set factor⟩

⟨set term⟩ ::= ⟨set factor⟩ | ⟨set factor⟩ + ⟨set term⟩

⟨set phrase⟩ ::= ⟨set term⟩ | ⟨set term⟩ - ⟨set phrase⟩

⟨structural expression⟩ ::= ⟨set phrase⟩ | ⟨if clause⟩ ⟨set phrase⟩
else ⟨structural expression⟩

⟨structural assignment⟩ ::= ⟨simple set⟩ := ⟨structural expression⟩ |
⟨procedure identifier⟩ := ⟨structural expression⟩

Creation of class members

A class member may have values assigned to any of its properties by the statement which creates it; these values override values given in the class declaration. Values assigned to a member's structure are indicated as in the following:

create. (triangle/L(area:=10, line.I, line.J, line.K));

This indicates that a triangle is to be constructed which consists of (i.e., whose structure is) the three lines, and whose area is defined to be 10. Any class member indicated in the structure may also be created, if its index is undefined, as in the following:

create. (intersection/K(line/L(point.I,point.J),line/M(point.I,point.N)));

This process may be continued to any degree of nesting; during execution the effect is to create the leftmost member whose structure has already been created, then to repeat, until all members have been created. Any of the members being created may have values assigned to any of its properties.

Syntax of create

⟨undefined index⟩ ::=/⟨variable⟩

⟨defined member⟩ ::= ⟨class name⟩ | ⟨defined member⟩ ⟨index⟩

⟨undefined member⟩ ::= ⟨defined member⟩ ⟨undefined index⟩ |
⟨undefined member⟩ ⟨undefined index⟩ | ⟨undefined member⟩ ⟨index⟩

⟨new member⟩ ::= ⟨undefined member⟩ | ⟨undefined member⟩ (⟨new

description list ⟩)

⟨ description⟩ ::= ⟨ set⟩ |⟨short property assignment⟩ | ⟨new member ⟩

⟨ new description list⟩ ::= ⟨ description⟩ | ⟨description⟩,

⟨new description list ⟩

⟨ creation ⟩ ::= create. (⟨ new member⟩)

Each of the ⟨ new member ⟩'s is created by the single statement.

Syntax of destroy

⟨destruction⟩ ::= destroy.(⟨set⟩)

Recall that sets which contain a destroyed set are also destroyed, and that references to a destroyed set from a contained set are deleted from the contained set's cover.

Manipulation of properties

A ⟨ property name ⟩ alone is used to indicate a property where the set or class member is known by context, that is, within a ⟨class declaration ⟩ or ⟨new description list⟩ or within a ⟨template⟩ (to be described later). Otherwise, the class member in parentheses follows the property name. Thus,

```
class box [1] (real area :=2.);
totalarea :=2. X area (box.1);
area (box.1):= .5 X totalarea;
```

A single identifier may be the name for properties of many different classes (as, for example, cover and structure) since the class member is always made clear either explicitly or from context.

Syntax of properties and variables

⟨variable⟩ ::= ⟨simple variable⟩ |⟨subscripted variable⟩ |⟨property variable ⟩

⟨property variable⟩ ::= ⟨property name⟩ |⟨property name⟩ (⟨simple set⟩)

Control:I

Templates

A ⟨template ⟩ is a description of a collection of sets. It consists of a list of ⟨set variables⟩'s (each one a ⟨ set name ⟩ followed by a slash, or an asterisk, an

asterisk followed by a slash and an ⟨ unsigned integer ⟩, or an ⟨ undefined member ⟩), ⟨ set ⟩'s, and ⟨ Boolean expression ⟩'s. The ⟨ set variable ⟩'s are assigned values such that the structural conditions implied by the ⟨ set ⟩'s and the Boolean conditions of the ⟨ Boolean expression ⟩'s are satisfied. For example,

triangle/L(area = 10∧right, line.I, line/K, line.J)

indicates a right triangle (or collection of right triangles) whose area is 10 and which consists of line.I, a line (or collection of lines) whose index is unknown, and line.J. No properties are specified for the lines (in particular, their structures are not specified).

If part of the structure of a set is irrelevant or unknown, an asterisk may be used in its place in a ⟨ template ⟩. The asterisk signifies that its place could be occupied by any string of symbols which represent structure (including the "empty" structure). Thus,

object/L (*, line.I, *)

specifies any "object" which contains line.I. One example of such an object is

object.1 (line.1, triangle.2 (line.2, line.2, line.I))

An asterisk followed by a slash and an ⟨ unsigned integer ⟩ is similar to a ⟨set name ⟩ followed by a slash; the structure which is assigned to it may be referred to elsewhere by an asterisk followed by a period and the ⟨ unsigned integer ⟩. Thus,

for each line/K (*/1, point.J, */2) do

begin create. (line/L (*.1, *.2)); destroy. (line.K) **end**;

creates a collection of new lines not containing point.J. For further details concerning the asterisk, see the discussion of the ⟨ CL for ⟩ in Chapter IV.

The functions "and", "or", and "not" may be used within a ⟨ template ⟩ to indicate, respectively, structural conditions which must be simultaneously satisfied, or are alternatives, or are forbidden. For example,

inside/L (or ([circle/K, triangle/M], [triangle/M, circle/K]))

specifies an object which consists of a circle "inside" a triangle, or a triangle "inside" a circle. For further details, see the discussion of the ⟨ CL for ⟩ in Chapter IV.

Recall that Boolean conditions may be inserted into a ⟨template⟩ by means of a ⟨standard test⟩. For example, if the following declarations are made

standard test active;

class triangle (real area, Boolean active);

then

triangle/L (area = 10)

and

triangle/L (area = 10 ∧active)

specify the same collection of members, but

triangle/L (area = 10∧¬ active)

specifies a disjoint collection. Refer to the save and restore procedures in Chapter III for a detailed example.

The CL for

The ⟨CL for⟩ is used to assign values to the ⟨set variable⟩'s which appear in a ⟨template⟩; structure is assigned to the ⟨set name⟩'s which are followed by slashes and to the asterisks, and values are assigned to the undefined subclass indicators and indexes of the ⟨undefined member⟩'s. Thus, the ⟨CL for⟩ is essentially a sophisticated search procedure. For example,

for each object/K (triangle/L (*, line.I,*),square/M (*, line.I,*)) do ⟨statement⟩

searches for any object which consists of a triangle and a square with line.I in common. Many additional examples may be found in the Go-Moku program.

In the example above, the each specified that all possible assignments of values to L, K, and M were to be made which satisfied the ⟨template⟩; first could have been used instead, with the obvious significance. If the first is used, then clearly the results may depend upon the order in which assignments are made. Note also that the ⟨statement⟩ could create, modify, or destroy class members; since the search locates one after another of the examples of the ⟨template⟩, the results of the for each may also depend upon this order.

A second type of ⟨CL for⟩ is used to assign each element, one after the

other, (or the **first** element, or the **last** element) from a specified ⟨set⟩ to a ⟨set name⟩. The process is simply one of renaming sets. Thus, for example, the statement

for each A **in** [B, C, D, E] **do** A:=A+F;

is equivalent to the sequence of statements

B:=B+F;

C:=C+F;

D:=D+F;

E:=E+F;

In both of the forms of the ⟨CL for⟩, as in the ALGOL ⟨for statement⟩, the ⟨statement⟩ following the **do** is not executed at all, if the conditions of the ⟨CL for⟩ cannot be satisfied. Thus, the ⟨CL for⟩ may be used to determine whether certain class members exist.

Syntax of the for statement

⟨for statement⟩ ::= ⟨for clause⟩ ⟨statement⟩ | ⟨label⟩: ⟨for statement⟩ | ⟨CL for⟩ ⟨statement⟩

⟨set variable⟩ ::= ⟨undefined member⟩ | ⟨set name⟩ / | * | */ ⟨unsigned integer⟩

⟨basic structure⟩ ::= ⟨set variable⟩ | ⟨set⟩

⟨template⟩ ::= ⟨basic structure⟩ | ⟨basic structure⟩ (⟨condition list⟩)

⟨condition item⟩ ::= ⟨Boolean expression⟩ | ⟨template⟩

⟨condition list⟩ ::= ⟨condition item⟩ | ⟨condition item⟩, ⟨condition list⟩

⟨for adjective⟩ ::= **each** | **first**

⟨set adjective⟩ ::= **each** | **first** | **last**

⟨CL for⟩ ::= **for** ⟨for adjective⟩ ⟨template⟩ **do** | **for** ⟨set adjective⟩ ⟨set name⟩ **in** ⟨set⟩ **do**

The Problem Solving Executive

Control:II

An invocation of the PSE is, as discussed earlier, similar to a ⟨CL for⟩, with

the important addition that the PSE can not only search for specified objects, but may also construct them. This construction is limited to the construction of members of goal classes from other class members. Thus, members of non-goal classes must be created by declaration or explicit create statements; this is necessary since the non-goal classes are the objects of the problem and are manipulated by the rules of the problem, rather than by the PSE. The goal classes, on the other hand, represent hypotheses about the problem, and hence are subject only to rules defined by the problem solver.

The desired goal members are described to the PSE by means of a ⟨template⟩ and/or members of one or more descriptive classes. For example,

for first subgoal/I (object.J, object.K) **during** time **do** ⟨statement⟩

specifies that a "subgoal" is to be constructed from the two "objects" if they exist and can be found within the time limit, "time". In this case, the ⟨template⟩ completely describes the desired subgoal, so no descriptive member is referenced. On the other hand,

for each subgoal/K (object.J, *) **during** time **do** ⟨statement⟩

incompletely specifies the desired subgoals: the structure implied by both the ⟨template⟩ and by some descriptive member must be satisfied by each subgoal. Finally,

for each subgoal/K **during** time **do** ⟨statement⟩

and

for each subgoal/K(value > bestval) **during** time **do** ⟨statement⟩

do not make any structural restrictions on the desired subgoals; structure must be supplied by descriptions which were created earlier.

If it is desirable that the goals be described by members of a specific class, or by a specific member, this can be indicated in the ⟨template⟩. For example.

for each description/K (subgoal/L) **during** time **do** ⟨statement⟩

or

for each description.I (subgoal/L) **during** time **do** ⟨statement⟩

In each of the above examples, time is initially set to the time allowed;

whenever control leaves the PSE, time is equal to the remaining amount of time. Thus, it is rather easy to devise quite complex ways of allocating search effort. For example, the following code searches for "moves" which will achieve "subgoals" of increasing value. After N moves have been found and placed in moveset, the remaining time is devoted to finding "traps" (presumably very high-value goals). This might be a section of a game player.

```
moveset:= empty; bestval:=0; M:=0;
for each subgoal/K (move/L,*,value > bestval) during time do
begin moveset:=[move.L] + moveset;
  bestval:= value (subgoal.K);
  M:=M+1;
  if M = N then go to exit end;
exit: for each trap/K (move/L, *, value > bestval) during time do
begin moveset:=[move.L]+ moveset;
  bestval:= value (trap.K)
end;
```

Syntax of the PSE statement

⟨ PSE statement ⟩ ::= **for** ⟨ for adjective ⟩ ⟨ template ⟩ **during** ⟨ variable ⟩ **do** ⟨ statement ⟩

Improvement

The basic operations of improvement in the Heuristic Programming System are the construction, destruction, and alteration of members of descriptive classes. The construction of a new descriptive member consists of the following steps: first, recognition of a new goal member; second, creation of a descriptive member whose structure consists of this goal; third, editing of the descriptive member to alter structure or properties; fourth, abstraction of the descriptive member.

Editing, as noted earlier, is a process by which a description is changed without changing the thing described. An ⟨ edit statement ⟩ consists of the name of the description and either a single ⟨ replacement ⟩ or a block of ⟨ replacement ⟩'s. Each ⟨ replacement ⟩ either assigns a new value to each instance in the description of a

property, or replaces each instance of a given ⟨template⟩ in the description by new structure and property values. A ⟨template⟩ is deleted from a description if it is replaced by **null**. Each ⟨replacement⟩ replaces part of the description by a reference to something which is strictly local to the description; in this way arbitrary replacements may be made without any effect upon any "external" objects, but it is still possible to refer to "local" objects by the names of the corresponding external objects.

Abstraction is a process of changing a (possibly edited) description into an "abstract" object, in which all indexes are undefined. All pointers to external objects are replaced by pointers to local blocks, which represent the objects and all their properties (including subclass indicators). Essentially, this is just the process of changing a class member into a ⟨template⟩ which contains only ⟨undefined member⟩'s.

The improvement process will be illustrated by a somewhat detailed example, the generation of a subgoal description in a tic-tac-toe program. Although the application is trivial, the idea behind this example is quite powerful; indeed, it forms the basis of the improvement section of the Go-Moku program of Chapter III.

Assume that the data structures of the program are "squares", "lines", "subgoals", and "descriptions". Each square has an integer-valued property called "occupant" with values "X", "O" or "unoccupied". Each line has an integer-valued property called "occupant" with values "X", "O", "unoccupied", or "blocked", and an integer-valued property "number" which may have any value from zero to three (zero if the line is blocked, otherwise the number of occupants). Each line consists of three squares. A description of a subgoal consists of an unoccupied square, a list of lines and the squares which they contain, and the properties of the lines and squares. Each subgoal has two properties, its "side" ("X" or "O") and its "subgoalvalue", with a value of $1/2^n$ for some n. The interpretation is that if the present configuration should contain the specified lines and squares and if the player with the proper side should occupy the square, then the resulting configuration would lead to three in a line in n moves or less regardless of the moves made by the

opponent (although if he has a more valuable subgoal, he might be able to achieve three in a line first, and therefore win).

Assume now that the opponent, X, has just moved to square.I and that his move has created two subgoals, subgoal.J and subgoal.K, with values $1/2^m$ and $1/2^n$, which cannot be simultaneously blocked. Evidently his previous move occupied the square of a subgoal with subgoalvalue equal to the minimum of $1/2^{m+1}$ and $1/2^{n+1}$. The problem is to create a description of this subgoal.

The first step has already been done: the relevant objects, square.I, subgoal.J, and subgoal.K, have been found. The second step is to create a description of these objects:

```
create.(description/L (subgoal/M (subgoalvalue:=if subgoalvalue
  (subgoal.I) > subgoalvalue (subgoal.J) then subgoalvalue (subgoal.J)/2.
  else subgoalvalue (subgoal.I)/2., side:=X, square.I, structure
  (subgoal.J), structure (subgoal.K))));
```

(Note the use of "structure": the subgoal will consist of lines and squares, not of other subgoals.) This object is a description of the situation after, rather than before, the move has been made. Thus, the occupant of square.I, and its effects, must be removed:

```
edit description.L do
begin occupant (square.I):=unoccupied; line/N (*/1, square.I,*/2):=
  line.N (number:=number (line.N)-1, *.1, square.I, *.2)
end;
```

Now the description is abstracted:

```
abstract (description.L);
```

This is a complete description of a subgoal which the opponent can achieve. A subgoal which the program can achieve is produced by the following:

```
create.(description/M(structure (description.L)));
edit description.M do
begin side (subgoal/N):=O;
```

```
        occupant (line/N):=if occupant (line.N) =X then O else if
            occupant (line.N) = O then X else occupant(line.N);
        occupant (square/N):= if occupant (square.N)=X then O else if
            occupant (square.N) = O then X else unoccupied
    end
```

Syntax of editing

```
    ⟨member⟩ ::= ⟨defined member⟩ | ⟨undefined member⟩
    ⟨property replacement⟩ ::= ⟨property name⟩ (⟨member⟩):=
        ⟨Boolean expression⟩ | ⟨property name⟩(⟨member⟩):=
        ⟨arithmetic expression⟩
    ⟨structural replacement⟩::= ⟨template⟩ :=null | ⟨template⟩:=
        ⟨new description list⟩
    ⟨replacement⟩::= ⟨property replacement⟩ | ⟨structural replacement⟩
    ⟨compound replacement⟩::= ⟨replacement⟩ | ⟨replacement⟩;
        ⟨compound replacement⟩
    ⟨edit block⟩ ::= ⟨replacement⟩ begin ⟨compound replacement⟩ end
    ⟨descriptive member⟩ ::= ⟨descriptor name⟩ ⟨index⟩ | ⟨descriptive
        member⟩ ⟨index⟩
    ⟨edit statement⟩ ::= edit ⟨descriptive member⟩ do ⟨edit block⟩
```

Syntax of statements

```
    ⟨statement⟩ ::= ⟨unconditional statement⟩ | ⟨conditional statement⟩ |
        ⟨for statement⟩ | ⟨PSE statement⟩ | ⟨creation⟩ | ⟨destruction⟩ |
        ⟨edit statement⟩
```

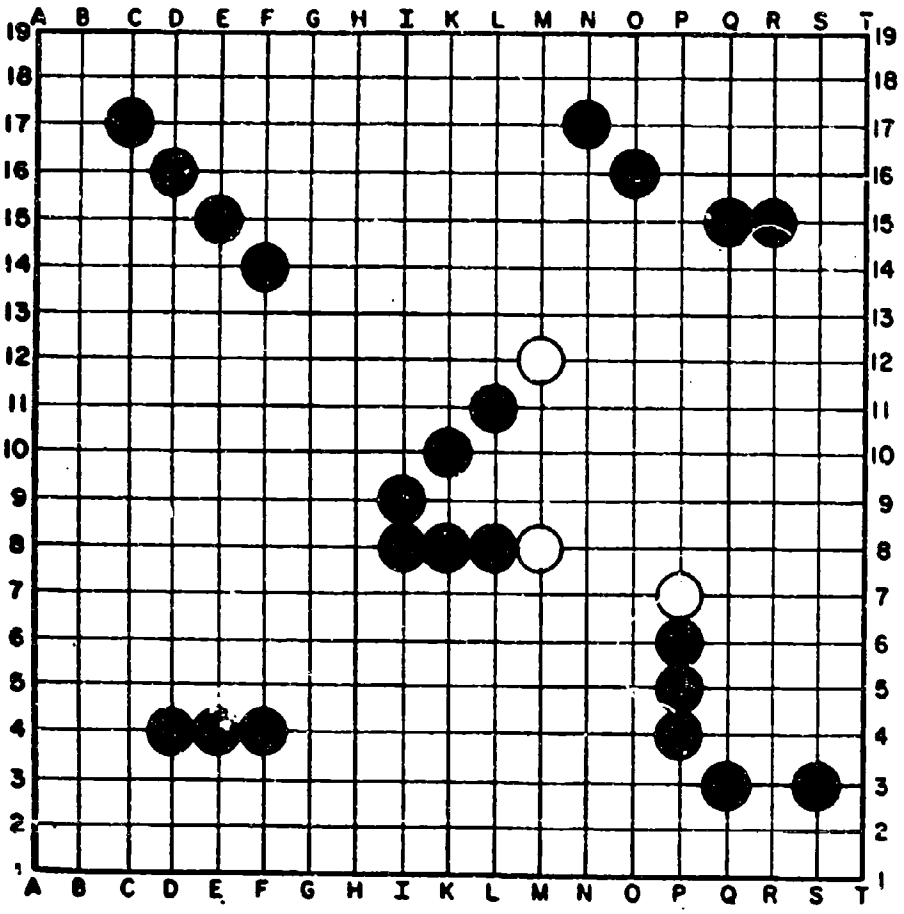
CHAPTER III
A SAMPLE PROGRAM

The Go-Moku program is a compromise between the conflicting objectives of simplicity, so that the program will be readily understandable, and complexity, so that it will be a significant demonstration of the power of the System. The program tends rather strongly toward complexity; this is more a reflection of the complexity of the ideas behind the program than of the difficulties in expressing those ideas. A brief description of Go-Moku is given below, followed by the description and text of the program. More details of Go-Moku may be found in Lasker (1960).

Description of Go-Moku

Go-Moku is played on a standard Go board, i.e., on the intersections of a 19 x 19 grid (see figure 1, in which many game segments are superimposed). Black and white pieces, called stones, are placed alternately until one player has attained a winning combination, which consists of *precisely* five stones in a row (horizontal, diagonal, or vertical). Pieces may not be moved, and all pieces are equal, which makes the game somewhat easier to program than chess or checkers. Certain elementary strategies are rather obvious. For example, even if white has the move in the game illustrated in the upper left, black has a certain win: if white occupies B18, then black occupies G13, and vice versa. Similarly, if black has the move in the lower left, he can force a win by occupying either C4 or G4; he is said to have produced an "open four" (i.e., four pieces in a line which can be extended in either direction to a winning "five") from the "three". A third simple example is shown in the middle of the diagram. Here, if black has the move, he can occupy H8; white cannot block both four's simultaneously, so black must win. A slightly more complicated example is shown in the lower right. If black occupies P3, creating a four-three, then white will be forced to occupy P2; black can then produce an open four by playing R3.

The game player tests for each of the above mentioned winning combinations in the evaluate procedure. The only complication which arises in the coding is due to the necessity for determining whether the four produced according to the final



VARIOUS GO-MOKU POSITIONS

Figure 1

example is, in fact, open; reference to the diagram and experimentation on a board or with pencil and paper will undoubtedly make the coding easier to follow.

One special rule of the game, which adds greatly to the complexity of the play, is illustrated in the upper right: two three's which could become open four's cannot be produced by a single play, unless that play is necessary to block the formation of a five on the opponent's next move. Thus, black cannot occupy P15. If white were to occupy P16, P17, P18, and P19, then it would be permissible for black to occupy P15. This rule is necessary in advanced playing, since otherwise black - the first to move - would very quickly be able to develop two-two combinations which would eventually become open four's.

Description of the Go-Moku Program

The simplest data structure in the game player is the point, i.e., the intersection of two grid lines. This is too simple a structure to manipulate directly; the line - five points, at least one of which is occupied, which could develop into a winning combination - is much more useful. Lines are, in turn, combined to form intersections - two lines which intersect in an empty point. Both lines and intersections are further classified according to the number of occupied points. For example, again referring to figure 1, (B4, C4, D4, E4, F4) and (C4, D4, E4, F4, G4) are both objects of type line.3; they form an intersection with the empty point C4. The intersection object is, then, described as intersection.3.3.I (C4, line.3.J (B4, C4, D4, E4, F4), line.3.K (C4, D4, E4, F4, G4)), where I, J, and K are the indexes of the objects within their classes, and the B4, C4, ... represent the appropriate points (actual coding would, of course, refer to point.L, for some L, rather than to the grid coordinates, but this would be rather lengthy and confusing in these examples). Similarly, in the lower right of the diagram, there is the object intersection.3.2.I (P3, line.3.J (P6, P5, P4, P3, P2), line.2.K (P3, Q3, R3, S3, T3)).

A subgoal consists of an unoccupied point and a number of lines and intersections; occupying the point will either produce a configuration which is, supposedly, desirable, or it will block the opponent's attainment of that configuration, depending the "side" of the subgoal. Descriptions of new subgoals are

produced whenever the opponent makes an unexpected move which results in the attainment of a favorable position for him.

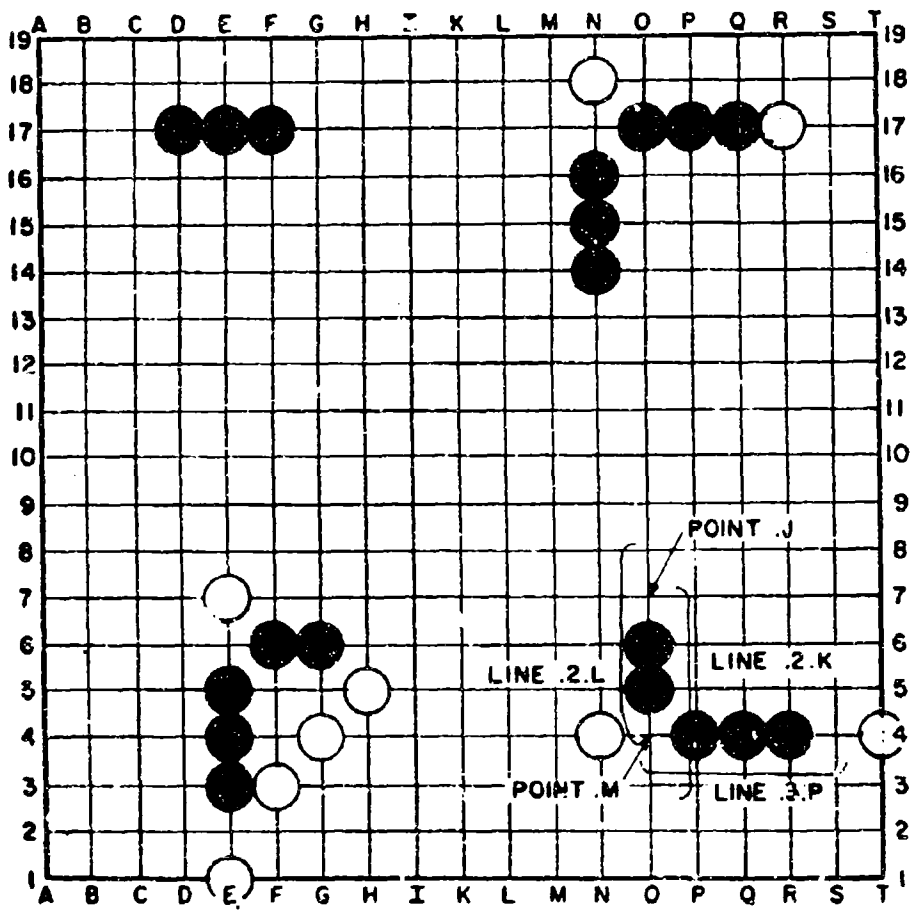
The major procedures of the program are analyze, which updates the data structure when a move is made, evaluate, which checks for the presence of certain obvious traps, generatemove, whch generates a set of plausible moves, and minimax, which produces a minimaxed value for a proposed move. The main program invokes generatemove to produce a set of moves, invokes minimax for each move to obtain its value, makes the best move, obtains the opponent's reply, and, if necessary, constructs a new subgoal description.

The generatemove procedure constructs subgoals, each of which indicates a plausible move and its estimated value. The minimax procedure either invokes generatemove to obtain a set of continuations or constructs subgoals to determine the value of a terminal position. Both generatemove and minimax are given time limits within which to do their searching and construction; this is necessary for realistic operation of the program, since subgoal descriptions can involve arbitrarily large numbers of lines and intersections, with many relations between them, so that the time involved in searching for possible instances of subgoals could be very large. The minimax procedure allocates time to the exploration of each continuation in proportion to its estimated value; the time spent in generating the move sets also decreases with increasing depth of exploration of the tree of possible moves. This has the following advantages: there is more chance of immediately recognizing a valuable but rare or complex position; searches further down in the tree can take advantage of subgoals constructed at earlier positions; the alpha-beta cutoff is more efficient since moves are more accurately ordered; and, finally, the most time is spent on those positions which are most likely to arise in the actual play of the game.

The goal of this program was, of course, to demonstrate the Heuristic Programming System, so a brief comment on its power is appropriate. First, the flexibility and efficiency of representation of board positions and subgoals is a major advantage of the System. The fact that lines and intersections can be represented as

blocks of pointers, rather than as lists, represents a storage savings of about fifty percent. The fact that complex entities, such as intersections, can be readily represented explicitly leads to great savings in time: the entities are created once, but each may be referred to many times while searching for subgoals. Koffman's program (1967) and Murray and Elcock's program (1968) involve complex subgoal descriptions, but each represents intersections implicitly (points and lines are the only explicit entities), so that, for example, searching for instances of the configuration in the lower right of figure 1 is a non-trivial operation. Second, the distribution of playing time within the program is very easily and flexibly controlled; the rather complex scheme discussed above was implemented by means of only a few lines of code. Other schemes might be substituted with very little effort. Third, the relatively simple implementation of improvement within the program is due in part to the facilities provided by the system, and in part to the representation. The method of improvement is similar to that used in Koffman's program (1967) and Elcock and Murray's programs (Elcock and Murray, 1967, and Murray and Elcock, 1968).

A fourth point involves the point that use of a minimax strategy greatly increases the power of a game player. The above mentioned programs suffer from the fact that the subgoals which determine their moves are descriptions of configurations for only one side (cf. Koffman, 1967, pp. 76-78). For example, in the configuration in the lower left in figure 2, assume that black, the program, is to play. If the move is chosen by considering each side's stones in isolation, black would, presumably, place a stone at E6; white would reply at E2 and win on the next move. Such a foolish move on the part of black would, of course, be avoided by even a novice human player - not because his list of subgoals is more complete, but because it is supplemented by minimax. Rather clearly it is impractical to make use of subgoals which include constraints on both sides; the subgoals would be far too numerous and complex. Therefore, the claim that minimax greatly increases the power of the program seems fairly reasonable. More to the present point is the fact that the minimax was very easily, yet efficiently, implemented using the facilities of



THE BEST LAID SCHEMES

Figure 2

the Heuristic Programming System. When a point is occupied, all class members which depended upon it being unoccupied have their "active" property made **false**; this property is checked in all templates, and "inactive" class members are rejected. When the point is again unoccupied (by backing up in the minimax) these members are made "active"; if it were not for the pointers from each object to the objects which contain it, this would be extremely time consuming.

The most interesting parts of the program are presented in the last few pages of coding, which involve the choice of moves and the generation of new subgoal descriptions. The coding of the rest of the program, with the possible exception of generatemove, is rather dull.

A final note on the storage requirements of the program: each move can create at most 20 lines and 1,200 intersections, which would require roughly 16,000 pointers; this is assuming many stones of the same side in the vicinity, and none of the opponent's stones. On a computer with 250,000 available pointers (an IBM 360 with 1,000,000 bytes, or a CDC 6600 with 131,000 words) this would allow a minimax up to 15 deep, which is quite adequate. (Actual moves tend to destroy roughly as many pointers as they create, so that the only space consuming operation is the minimax, which can only inactivate structures.)

The Go-Moku Program

begin comment *the complete Go-Moku program, except for input/output routines (input, nextmove, displaymove, victory, defeat, tie), is given below;*

class line [4,] (**integer** direction:=direc,ct:=0, occupant:=present, structure:=5,**Boolean** active:=**true**),

point [441] (**integer** occupant:=empty, ct:=0,structure:=0, **real** subgoalvalue, **Boolean** active:=**true**);

unordered class intersection [4,4,] (**integer** structure:=3, ct:=0, owner:=present, **Boolean** parallel, active:=**true**);

goal class subgoal (**integer** side, ct:=0, **Boolean** backedup:=**false**, active:=**true**);

descriptive class description;

standard test active;

set tempset, set, pushdown;

rankhi by subgoalvalue **set** moveset;

integer I, J, K, L, M, N, P, Q, base, empty, mine, his, border, present, other, direc, west, nw, north, ne, maxdepth, maxbreadth;

real R, S, T, U, V, alpha, beta, C, maxcost, mintime, learningcost, testcost;

switch S1:=force, force, lose, select;

comment *the parameters maxdepth, maxbreadth, C, maxcost and mintime control the minimax procedure by determining, respectively, the maximum depth of search (forced moves are not counted), the maximum breadth, the fraction of the available time to be spent on the generation of moves, the maximum amount of time for each move, and the minimum amount of time during which further exploration is feasible. The parameters learningcost and testcost determine the amount of time to be spent after each move in analyzing the opponent's move and, if necessary, constructing a new subgoal. All the preceding parameters are read in by the input procedure.;*

Boolean procedure legal (move, play);

value move, play; **integer** move, play;

begin integer I;

```
comment this procedure returns the value true if move is a legal move for the
player whose side is play. A move is illegal if and only if it would create two
threes, each of which could become an open four, and it does not block the
formation of a five on the opponent's next move.;
legal:= true;
for first line.4/I(occupant≠play,*,point.move,*) do go to exit;
for first intersection.2.2/I(owner=play∧¬ parallel,point.move,*)do
legal:=false;
exit: end legal;

integer procedure evaluate (M); integer M;
begin integer I, J, K, L, N, P, Q, R, S, T;
comment evaluate will determine whether a forced win is possible, or a blocking
move is necessary, or a loss is inevitable, or if there is no obviously forced move.
The value returned will be 1, 2, 3, or 4, respectively. M will be set equal to the
index of the forced move, if there is one. First evaluate checks to see if there is a
four.;
evaluate:=4;
for first line.4/I (side = present,*, point/M (occupant=empty),*) do
begin evaluate:=1; go to exit end;
comment now see if he has any immediate wins.;
for first line.4/I (*,point/M(occupant=empty),*) do
begin evaluate:=2;
comment see if he has other fours.;
    for first line.4/J (*,point/K(occupant=empty),*,M≠ K) do evaluate:=3;
go to exit
end;
comment now see if a win can be forced on the next move. This is done by first
searching for intersections with three occupied points. If the lines in such an
intersection are in the same direction then occupying the empty point must produce
an open four. If the lines are not in the same direction, then occupying the empty
```

point must produce two fours. See the upper right and upper left examples in figure 2.;

for each intersection.3.3/I (owner = **present**, point/M,°) **do if legal** (M, present) **then**
begin evaluate:=1; **go to** exit **end**;
comment *now block any such move that he might make;*
for each intersection.3.3/I (point/M,*) **do if** legal (M, other) **then**
begin evaluate:=2; **go to** exit **end**;
comment *the final forced win to be considered involves a three-two intersection. If the lines are not in the same direction, then occupying the intersection will produce a four and a three. If, in blocking the four, the opponent does not create a four of his own, then the three can become another four which will lead to a win if it is open. Hence, two conditions are necessary for a forced win: the opponent's move does not create an immediate threat, and the four which is ultimately produced must be open. A preceding comment dealt with the second condition: an open four can be produced by occupying the intersection of two three's in the same direction. Hence, if there are two two's in the same direction which intersect a three in one point and each other in another point, then the second condition is satisfied. See the example in the lower right of figure 2.;*
for each intersection.2.2/I (owner=presentΛparallel,point/J, line.2/K, line.2/L) **do**
for each intersection.3.2/N (point/M, line.3/P (*, point/Q (occupant=empty),
*, point/R (occupant=empty), *), line.2.K, J≠M) **do**
begin for first intersection.3.2/S (point.M, line.3.P, line.2.L) **do**
    begin comment *check to see if he will have an immediate threat.;*
    for first line.3/T (occupant=his, *, or ([point.Q], [point.R]), *) **do**
    go to next
    end;
evaluate:=1; **go to** exit;
next: **end**;
exit: **end** evaluate;

Boolean procedure analyze;

comment *base is the index of the position which has just been occupied by a present piece (where present is determined outside the procedure to be either mine or his). The procedure saves all classes within which point.base previously appeared and then, if the game is not over, finds all sets of four points which can be combined with base to form lines of one, two, three, or four points. A check is made for the edge of the board and, because a winning combination is precisely five in a row, a check is made to ensure that there is not an adjoining piece which would form a six. Analyze constructs all the intersections containing the newly created lines, where an intersection is defined to be a class consisting of an unoccupied point and two distinct lines which contain it. The lines may have other points in common, in which case parallel is* **true**. *Analyze has the value* **true** *if a five in a row has been found (in this case the lines and intersections are not created), and the value* **false** *otherwise.;*

begin integer I, J, K, L, M, count; **set** tempset;

comment *save the cover (i.e., all the containing objects) of point.base. Refer to the save procedure for a detailed explanantion.;*

save;

occupant (point.base):=present;

comment *iterate over the four directions in which straight lines can be formed;*

for direc:=west, nw, north, ne **do**

begin comment *now iterate over the five lines through point.base in the current direction;*

J:=base;

for I:=1 **step** 1 **until** 5 **do**

begin if occupant (point.(J-direc))≠ **present** ∧ occupant (point.(J + 5 × direc)) ≠ present **then**

begin comment *the ends of the current line are not occupied by present, so a winning line might be possible. .Now look at each point of the*

```
    five in the line;
    tempset:= empty;
    count:=0;
    K:=J;
    for L:=1 step 1 until 5 do
        beginM:=occupant(point.K); if M=presentthen count:=count + 1
        else if M ≠ empty then
        begin comment current point is either an edge or an other piece.
        Go to L2 if all the other lines will include this point, otherwise
        go to L1;
        if L > 1
        then go to L1
        else go to L2
    end;
    tempset:=tempset + {point.K};
    comment add to set of points and go to next point;
    K:=K + direc
end L;
if count=5 then begin analyze:=true; go to finish end ;
        create.(line.count/L(structure(tempset)));
        comment now see if any of the points in the line are in other
        lines. If so, create new classes which represent intersections of
        lines in unoccupied points;
        for each tempset (*, point/M(occupant=empty),*) do
            begin integer N, P, Q;
                for each line/N/P (P≠L,occupant=present,*,point.M,*) do
            begin comment create a new intersection. The first two
                indexes represent the number of occupied points in each
                of the two directions. The first component is the point
                at the intersection, the next two are the intersecting
```

```
                    lines. If the lines are in the same direction, parallel is
                    true.;
                        if N > count then
                        create.(intersection.N.count/Q(parallel:=
                        direction(line.count.L)=direction(line.N.P),
                        point.M, line.N.P, line.count.L))
                        else
                        create.(intersection.count.N/Q(parallel:=
                        direction(line.count.L)=direction(line.N.P),
                        point.M, line.count.L, line.N.P))
                    end creation of intersection
                end current intersecting line
            end current line, so shift line in current direction.;
            L1: J:=J - direc;
        end I;
L2 : end direction;
analyze:=false;
finish:end analyze;
procedure save;
begin set A, temp; integer I, J;
comment save will inactivate the data structures which depend upon point.base being
unoccupied. A slight complication is introduced by the rule that a winning line
consist of precisely five stones in a row - it is necessary to inactivate all lines
occupied by the present player which would become sixes if point.base were
occupied. This is accomplished by adding these lines to the cover of point.base, so
that they will be properly reactivated by restore.;
for I:=west, nw, north, ne do
    for each line/J(owner=present, direction=I, or ([*.point.(base+I)],
    [point.(base-I),*])) do cover (point.base):=cover (point.base) + [line.J];
```

```
comment save point.base in pushdown so that its cover may be retrieved by
restore.;
pushdown:=pushdown + [point.base];
comment now inactivate the cover, to prevent the objects containing point.base from
satisfying a template. This amounts to the temporary destruction of the objects. The
reference count, ct, is increased, so that ct will be the number of times each object
was inactivated. It must be restored the same number of times to again become
active.;
temp:=cover (point.base);
for each A in temp do
begin temp:=temp + cover (A) - [A];
  active (A):=false;
  ct(A):=ct (A) + 1 end
  end save;

procedure restore;
begin set A, temp; integer I;
comment restore will remove the last point from pushdown, destroy the active
objects which include it, and then reactivate the structures which it inactivated, thus
essentially reversing the most recent save operation.;
for first pushdown (*, point/I) do
pushdown:=pushdown - [point.I];
temp:=cover (point.I);
for each temp(*/I, A/ ,*) do destroy.(A);
for each temp(*/I, A/ (¬active),*) do
begin temp:=temp + cover (A) - [A];
  ct (A):=ct(A)-1;
  if ct (A) = 0 then active (A):=true end
end restore;
```

```
procedure generatemove (limit, set, number, total);
rankhi by subgoalvalue set set;
real limit, total; integer number;
begin comment generatemove finds moves and places them in set. The number of
moves is returned as number, and the sum of their subgoalvalues as total. The
available search time, limit, is divided among five ranges of values of subgoals. Each
subgoal has the value assigned to it when it was described, or a value determined
by minimax. In the latter case, backedup is true. The subgoalvalue of a move is
either the value of the subgoal within which it appears (if the value is minimax or
if the move appears in no other subgoals), or the sum of the values of the two
most valuable subgoals within which it appears. This latter process gives a better
estimate of the value of a dual purpose move than simply choosing the highest
value: e.g., if two subgoals are each k moves from victory, and hence have value
1/2^k, and cannot be simultaneously blocked, then together they force a position k-1
moves from victory, with value 1/2^k + 1/2^k = 1/2^(k-1);
integer I, J, K; Boolean duplicatefound;
real time, maxtime, minimum;
rankhi set tempset;
number:=0; total:=0.;
time:=limit ×.2;
for minimum:=.0001,.001,.01,.05 do
    begin comment look for subgoals with values at least minimum.;
    maxtime:=time;
    for each subgoal/I (point/J, *, value ≥ minimum) during maxtime do
        begin if legal (J, present) then
        tempset:=[subgoal.I] + tempset end;
    limit:=limit - time + maxtime
    end;
comment generate less valuable goals, until either time runs out or the number of
moves becomes large.;
```

```
for each subgoal/I (point/J, *, value < .0001, number < 2 × maxbreadth) during
    limit do
      if legal (J, present) then
    begin number:=number + 1; tempset:=tempset + [subgoal.I] end;
comment now compute the subgoalvalue of each point and put the point in set.;
set:= empty; number:=0;
for each tempset (subgoal/I (point/J,*), *, number < maxbreadth) do
  begin duplicatefound:=false;
findval: tempset:=tempset - [subgoal.I];
    subgoalvalue (point.J):=value (subgoal.I);
    if backedup (subgoal.I) then
        for each tempset (*, subgoal/K (point.J, *), *) do destroy. (subgoal.K)
    else for each tempset (*, subgoal/I (point.J, *), *) do
      begin if backedup (subgoal.I) then
        go to findval
        else if ¬duplicatefound then
        begin duplicatefound:= true;
          subgoalvalue (point.J):=subgoalvalue (point.J) + value(subgoal.I)
        end,
        destroy. (subgoal.I)
      end;
    set:=set + [point.J];
    number:=number + 1;
    total:=total + subgoalvalue (point.J)
  end
end generatemove;

procedure minimax (bestval, depth, cost);
value depth, cost, real bestval, cost; integer depth;
begin comment minimax is a minimax procedure modified by alpha-beta cutoff (see
discussion of search). The value of the move which occupies point.base will be
returned as bestval The maximum depth and cost are depth and cost, respectively.;
```

```
rankhi by subgoalvalue set moveset;
integer I, J, K, L; real Q, R, newcost;
switch S1:=win, block, lose, more;
comment save the index of the move.;
L:=base;
comment make the move and switch sides.;
analyze;
I:=present; present:=other; other:=I;
comment see if the next move is forced.;
go to S1 [evaluate (base)];
win: bestval:=if present = mine then 1. else 0.;
go to exit;
lose:bestval:=if present = mine then 0. else 1.;
go to exit,
comment if the next move is a forced block, then get the value of the best reply.
Note that depth is not decremented.;
block:minimax (bestval, depth, cost); go to exit;
comment the next move is not forced, so check to see if the maximum depth has
been attained or if there is insufficient time to explore more deeply.;
more:if depth ≠ 0 ∧ cost >mintime then
begin comment generate a set of feasible moves. The global parameter C determines
the fraction of the total time to be spent in generating the moves. I is the number
of moves generated. Q is the sum of the values of the moves.;
R:=cost × C;
generatemove (R, moveset, I, Q);
comment compute the time remaining divided by the total value of the generated
moves.;
R:=(cost × (1.-C))/Q;
comment now determine the minimax value of each generated move, allocating time
to each move in proportion to its value.;
```

```
if I = 0 then bestval:= 0. else
    begin if present = mine then
        begin bestval:=0.;
            for each moveset (*, point/base,*) do
            begin minimax (Q, depth-1, R X subgoalvalue (point.base));
                if Q > bestval then bestval:=Q;
                if Q ≥ beta then go to exit
            end;
            beta:=bestval
        end
        else
        begin bestval:=1.;
            for each moveset (*, point/base,*) do
            begin minimax (Q, depth - 1, R X subgoalvalue (point.base));
                if Q < bestval then bestval:=Q;
                if Q ≤ alpha then go to exit
            end;
            alpha:=bestval
        end
    end
end
else
begin comment the maximum depth has been attained, so evaluate the present
position.:
generatemove (cost, moveset, I, Q);
if I = 0 then bestval:=.5 else
    begin for first moveset (point/I,*) do
        bestval:=subgoalvalue (point.I);
        comment if there are two subgoals of equal value but opposite side,
```

clearly the one belonging to the side with the move is the one which should determine the value of the position. The following code ensures this.;

```
        moveset:=moveset - {point.I};
check:  for first subgoal/J (point.I,*) do
        K:=side (subgoal.J);
        if K ≠ present then
            for first moveset (point/I,*) do
            if subgoalvalue (point.I) = bestval then go to check;
        comment the value now may vary from 0. to 1.. It must be adjusted to
    indicate the value to the program's side.;
        bestval:=if K = his then
        .5 - bestval × .5 else
        .5 + bestval × .5
    end
end evaluation for depth = 0.;
comment now the minimaxed value has been determined. Restore the previous board
configuration and side.;
exit:   restore;
I:=present; present:=other; other:=I;
comment attach the minimax value to the subgoal which contains point.L, so that
the improved value may be used if this subgoal is referred to later.;
for first subgoal/I (point.L, *) do
        value (subgoal.I):=abs(2. × bestval - 1.);
        backedup (subgoal.I):=true
end minimax;

comment this is the initialization section of the player. It creates the initial board
configuration of occupied points, sets various constants, and then goes to the move
section to make the choice of move. The board is considered to be 21 × 21,
which includes a border of points occupied by a special piece. Points are numbered
```

from 1 to 441, starting from the southwest (lower left) corner and increasing from left to right and bottom to top. First read in the parameters maxdepth, maxbreadth, C, maxcost, mintime, learningcost, and testcost, and initial subgoal descriptions.;

```
input;
comment set the directions of possible lines;
west:=-1;nw:=20;north:=21;ne:=22;
empty:=0;comment point unoccupied;
mine:=1;comment I occupy it;
his:=2;comment the opponent occupies it;
border:=3; comment off the board;
comment set the borders (interior initialized to empty by declaration);
for I:=1 step 1 until 21 do occupant (point.I):=border;
for I:=421 step 1 until 441 do occupant (point.I):=border;
for I:=22 step 21 until 400 do occupant (point.I):=border;
for I:=42 step 21 until 420 do occupant (point.I):=border;
comment nextmove will obtain the next move in the initial configuration. It will set
base to the index of the point, present to the side that moved, other to the
opposite side, and return with the value true. If there are no moves, it will return
with the value false, present will be set to the side which moves next, and other
will be set to the opposite side.;
next:    if nextmove then
    begin analyze; displaymove;
    comment analyze will create a data structure representing the lines and
    intersections which are formed by the move just made. Displaymove will display
    the move.;
    go to next
    end;
comment at this point the initial data structures - i.e., the lines which have at least
one occupant and which may develop into winning combinations, and the
```

intersections of these lines on empty points - have been constructed. The following code plays the game.;

```
if present = his then go to hismove;
comment evaluate returns a 1 if a win is certain, a 2 if a blocking move is
necessary, a 3 if a loss is certain, and a 4 otherwise. The switch goes to force,
force, lose, and select, respectively. If a move is forced, base will be set to its
index.;
loop:  go to S1 [evaluate (base)];
comment invoke defeat if a loss is inevitable.;
lose:  defeat;
comment now generate feasible moves. Procedure generatemove returns the moves in
moveset, and their number in K. R is the time limit and T is the sum of the
values of the moves.;
select:  R:=maxcost × C;
generatemove (R, moveset, K, T);
if K = 0 then tie;
comment now use minimax to choose the best move. The time to be spent on
evaluating each move, T, is proportional to its estimated value. The search depth is
maxdepth. The minimax value after occupying each point is returned as V. V
represents the value of the position to the program, with 0. as worst (i.e., he has a
sure win) and 1. as best (the program has a sure win). A subgoal value for a
position is the value for the side which can, in one move, achieve the subgoal. If
the most valuable subgoal has the value SV then V:=.5 - .5 × SV if he achieves it
and V:=.5 + .5 ×SV if the program achieves it. Alpha and beta are the cutoff
values (see discussion of minimax in Chapter 1).;
T:=(maxcost × (1.-C))/T;
alpha:=0.; beta:=1.; R:=0.;
for each moveset (*, point/base, *) do
    begin M:=base;
        minimax (V, maxdepth, T × subgoalvalue (point.base));
```

```
    if V > R then begin R:=V; K:=M end
    end;
base:=K;
comment display the move. The procedure uses base and present to determine
the point and side.;
force: displaymove;
comment now update the data structure and invoke victory if the game is
over.;
if analyze then victory;
comment get his reply and display it;
hismove: if ¬ nextmove then tie;
displaymove;
comment generate a list of his expected replies.;
R:=learningcost;
generatemove (R, moveset, K, T);
comment see if the actual reply was expected.;
for first moveset (*, point.base, *) do
    begin comment yes, so prepare for the next move.;
        go to cleanup
    end;
comment his reply was unexpected, so decide whether he is about to achieve
an unforeseen subgoal. This will be the case if, in making this move, he
prepares for two subgoals, both of which cannot be avoided by the program.
Otherwise, his move was either a mistake or was part of a subgoal which
cannot be foreseen even after his move. In the latter case, the subgoal will be
discovered later.
First, make his move.;
analyze;
present:=mine;
other:=his;
R:=testcost;
generatemove (R, moveset, K, T);
```

```
comment now find the two subgoals, if they exist.;
J:=0;
for each subgoal/I (side = his, *, point.base, *) do
    if J ≠ 0 then go to both else J:=I;
comment if control reaches this point, the two subgoals could not be found, so go
on to the next move.;
go to release;
comment two subgoals containing point.base were found, with indexes I and J,
which were evidently the components of the new subgoal which must now be
described. To visualize the present situation and subsequent operations, consider the
central configuration in figure 1. There are two three's which intersect at H8.
Assume that the only subgoal description consists simply of a four. Then, if the
opponent is black and he has moved to H8 (i.e., point.base is H8), the program
finds that his move was not on the list of generated moves, but two subgoals have
now been formed (i.e., the two four's). The subgoals are subgoal.I (line.4.X) and
subgoal.J (line.4.Y), for suitable X and Y (ignoring points for the sake of brevity).
The objective is to create a subgoal consisting of the intersection of two three's.
The first operation is to create an intersection for each line of subgoal.I which
intersects a line of subgoal.J in point.base. Note that point.base will be unoccupied
in the completed subgoal description, so that lines consisting of point.base and four
unoccupied points must be deleted.;
both:   tempset:=empty;
for each subgoal.I (*,line/K/L(*,point.base, *), *, K > 1) do
    for each subgoal.J (*, line/M/N(*,point.base, *), *, M > 1, L ≠ N ∪ K ≠ M)
    do
        if K ≥ M then
        begin create. (intersection.K.M/P ( point.base, line.K.L, line.M.N ,
            parallel:=direction (line.K.L)=direction (line.M.N))):
          tempset:=tempset + {intersection.K.M.P}
    end
```

```
else
begin create. (intersection.M.K/P (point.base, line.M.N, line.X.L,
parallel:=direction (line.K.L)=direction (line.M.N)));
    tempset:=tempset + {intersection.M.K.P}
end;
```

comment *to continue the example, intersection.4.4.P (point.base, line.4.X, line.4.Y) has been created and put into tempset. Now create the new description.;*

```
occupant (point.base):=empty;
create. (description/Q (subgoal/N (side:=his, point.base, structure (subgoal.I), structure
    (subgoal.J), structure (tempset), value:=if value (subgoal.I) < value (subgoal.J)
    then value (subgoal.I) X.5 else value (subgoal.J) X.5)));
```

comment *the result now is description.Q (subgoal.N (point.base, line.4.X, line.4.Y, intersection.4.4.P (point.base, line.4.X, line.4.Y))). Now the description must be edited to reflect the fact that, in the desired description, point.base will be unoccupied.;*

```
edit description.Q do
    begin comment fix up intersections.;
        intersection/K/L/M (*/1, line /L/J (*, point. base, *), L > 1):=
        intersection.K.(L-1).M(*.1,line (L-1).J;
        intersection/K/L/M(*/1,line/K/J (*,point.base, *),*/2,K>L):=
        intersection.(K-1).L.M(*.1,line.(K-1).J,*.2);
        intersection/K/L/M(*/1,line/K/J(*,point.base,*),*/2,K>1):=
        intersection.L.(K-1).M(*.1,*.2, line.(K-1).J));
        intersection/K/L/M(*,point.base,*):=null;
        comment now delete extraneous lines.;
        line/K/L(*,point.base,*):=null;
        comment the example is now description.Q (subgoal.N (point.base,
    intersection.4.4.P (point.base, line.3.X, line.3.Y))), with all other points removed
    from the description.;
```

```
    end;
comment abstract the description. Copy it, then use the original for a description of
his subgoal and the copy for a description of mine.;
abstract (description.Q);
create. (description/I(structure (description.Q)));
edit description.I do
    begin
            owner (intersection/J/K/L):=mine;
            occupant (line/J/K):=mine
    end;
comment everything has been done except for destroying temporary structures;
 for each set in tempset do destroy.(set);
occupant (point.base):=his;
go to release;
comment now update the data structure for his move.;
cleanup:  analyze;
present:=mine; other:=his;
comment the structures which were inactivated by analyze will no longer be needed.
so release their storage.;
release:  for each set in pushdown do destroy.(set);
go to loop;
end Go-Moku program
```

CHAPTER IV

THE HEURISTIC PROGRAMMING SYSTEM:II

Chapter II presented a general description of the System; this Chapter contains more detailed information, which might be of use in program optimization, or in the implementation or alteration of the System. Since this is intended for the knowledgeable user, the presentation is fairly brief.

Data Structures and Data Management

The ideas presented in this section are applications of the facilities of a general system described by Ross (1967b).

Recall from Chapter I that there are two types of structures within the System: blocks of contiguous storage of various sizes (representing class headers, class members, and set headers), and simple lists linking the blocks together. In addition, there are blocks, similar to class members, representing the results of the "and", "or", and "not" functions. Blocks have fixed structures which are determined at compilation time. List elements are simple in structure: each consists of a flag telling whether the element is the last of the list, a pointer to the next element (if the element is not last) or to the block containing the list (if the element is last), and a pointer to a block. Note that lists do not point to other lists; thus, there is never any problem in determining whether a list element can be returned to free storage. Neither a "garbage collection" routine nor a reference counter is necessary in the System, which greatly simplifies Data Management (*ibid.*, p. 485). List elements are obtained from, or returned to, a number of "zones" (blocks of contiguous storage) containing only list elements, as in the "SPEC" strategy of Ross (*ibid.*). This increases the speed of Data Management without causing a serious problem of storage fragmentation. Blocks of various sizes are obtained from a number of zones containing lists of blocks of various sizes, as in the "REG" strategy of Ross (*ibid.*). This tends to minimize storage fragmentation at some cost in time. However, list elements will generally be obtained from, or returned to, free storage much more often than blocks will be created or destroyed, so that the time

spent should not be significant.

In the event that free storage has been exhausted, the Data Management Routine invokes the reclamation procedure. This procedure aborts the program with an error message; the user may substitute his own procedure and take any desired action. For example, the procedure might destroy all subgoal members and descriptive members with value less than some constant. For further details on reclamation, see the section on system procedures.

More About the Problem Solving Executive

The PSE and the first form of the CL for (involving the template) are exceedingly complex search mechanisms, which are implemented via a single procedure with multiple entry points, as shown in the flowcharts of figures 3-6. The purpose of the procedure is to find an object (or collection of objects) matching a template. The template may be produced by the compiler, or it may be a descriptive member, or it may be produced by the conjunction of a compiler-produced template and a descriptive member (cf. the discussion of the PSE in Chapter II). In any case, a template is simply a description of a collection of objects; it consists of linked blocks, each containing a description of some element. The search procedure matches an object to the template simply by traveling through the blocks and the structure of the object, assigning values to set variables and checking to see that each condition is satisfied.

Du the following description of the search procedure, the object and template will sometimes be referred to as if they were in their original one-dimensional string representation, and sometimes as if they were in two-dimensional tree representation. There should be no confusion as to what is meant.

The general plan of the search is to proceed from the left end (of the string representation) of the template to the right end. Each time a decision has to be made (e.g., what index to assign to an undefined member, or what value to assign to an asterisk), pointers to the decision point in the template are put into two

stacks, called "pushdown" and "set" in the flowcharts. Each time a mismatch is discovered, the search begins again from the topmost decision point in pushdown, or, if there is none, the search procedure exits with an indication of failure. Pushdown is local to the search procedure, which calls itself recursively. Set is external to the procedure, so that when a match is finally found, it contains the location of all the decisions that were made; set is used by the foreach to make all the other possible decisions.

The procedure is reasonably straightforward; the flowcharts appear complex because of the many different cases which arise. Set names followed by slashes, or asterisks followed by slashes, or unpaired asterisks, are initially assigned a null structure, which is then increased by one set after another. Thus, given the template

A/K (*,B/L)

and the object

A.I (C.J (B.M), B.K)

the asterisk is initially assigned the empty string, which leads to a mismatch between B/L and C.J. Then C.J is assigned to the asterisk, and K to L, which is successful. Paired asterisks - i.e., pairs such that parentheses between them are balanced - are treated differently, since they may represent strings which are not parenthetically balanced. Thus, given the template

A/K (*, B/L, *)

and the object

A.I (C.J (B.M), B.K)

the result is to eventually assign "C.J(" to the first asterisk, "),B.K" to the second, and M to L. Thus, the left member of a pair of asterisks is assigned progressively longer strings, *without* regard to parenthesis level; the earlier mentioned set variables are assigned strings *with* regard to parenthesis level. Undefined members are assigned indexes and/or subclass indicators as shown above.

A small complication arises if the PSE has been invoked and an undefined member which is a subgoal is encountered in the template. If a suitable subgoal has previously been constructed, it is assigned to the undefined subgoal. If not, then a

subgoal must be constructed - i.e., a class member must be created and must be assigned structure. Structure can be assigned only if there is a complete description of it; hence, it may be necessary either to refer to some descriptive member or to form the conjunction of a descriptive member and the template. In the latter case, the search procedure is called to ensure that the result will not impose any contradictory conditions upon the subgoal being constructed. Descriptive members are chosen with the least costly first, and so on. Ties are resolved by choosing the most valuable first. Remaining ties are resolved by choosing the lowest subclass and index first. (Non-descriptive members are chosen starting from the lowest subclass and index.) Cost is determined from the complexity of the structure; value must be assigned by the user's program.

The "or" function produces a block of a special type, which informs the search procedure that a decision point must be put in the stacks. The "or" function is invoked whenever unordered sets are referred to in a template. The "and" function also produces a special block, which informs the search procedure that the next element in the object must satisfy two conditions. The "not" function establishes a condition which must not be satisfied. Note that these functions, and all others, are evaluated before the search begins.

The compiler arranges the Boolean conditions so that each is evaluated as soon as possible. This tends to save time; it also means that there may be conditions within the template which can be evaluated before search begins, which is occasionally quite convenient.

As an aid in reading the flowcharts, note the following remarks. The arguments to the COMPSTRUCT (compare structure) procedure are pointers to structure lists of the template (PT) and object (PO). The notation "OBJECT (pointer)" is occasionally used to refer to the object pointed to by the pointer; "STRUCTURE (OBJECT)" is the object's structure. The abbreviations "OBJ" and "STR" are also used. "SUCCESSOR" refers to the next element on the list currently being followed. Finally, note the legend at the bottom of the first flowchart, figure 3.

FIGURE 3. MAIN FLOW AND LEGEND

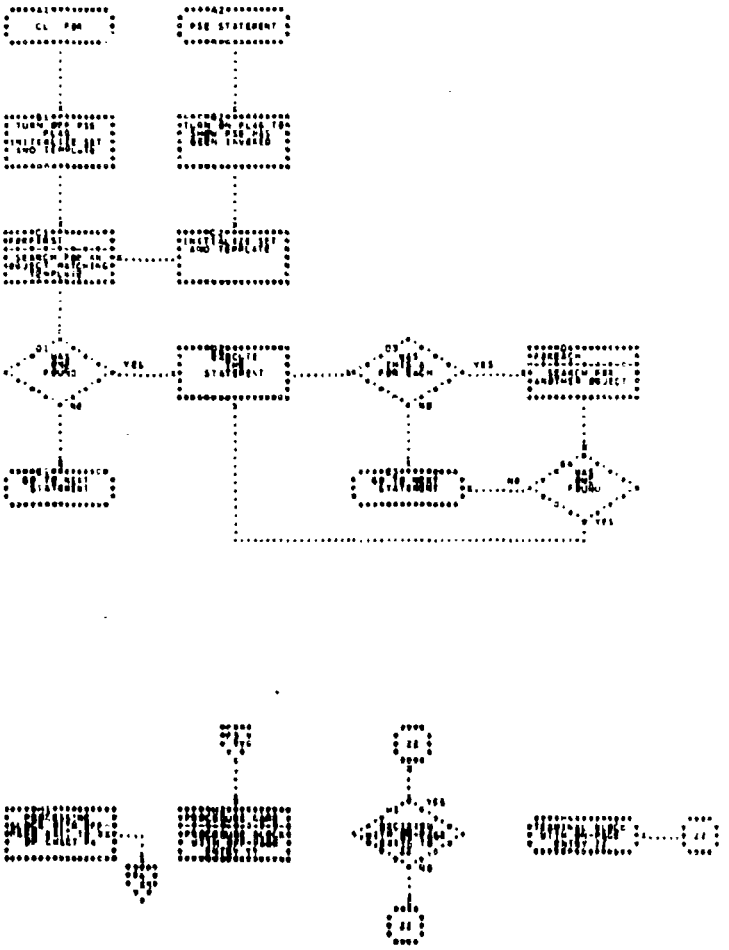


FIGURE 4. DETAILED FLOW

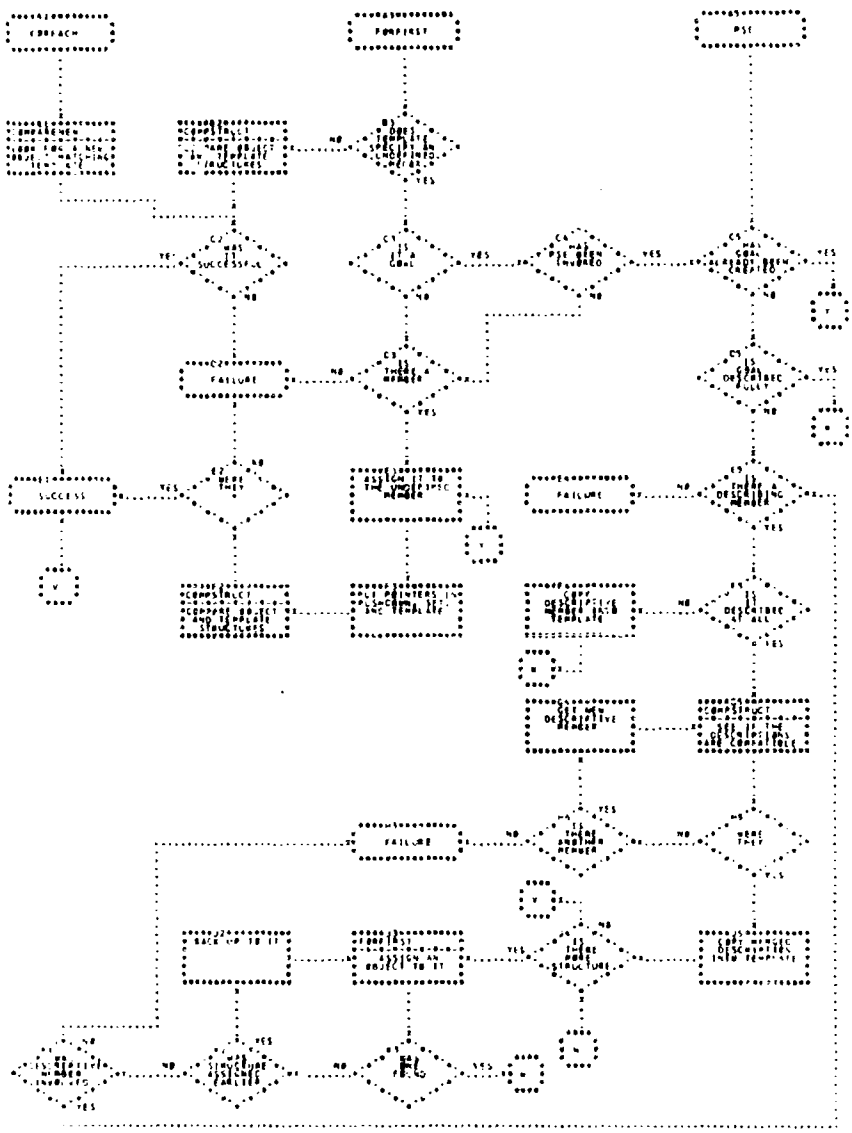
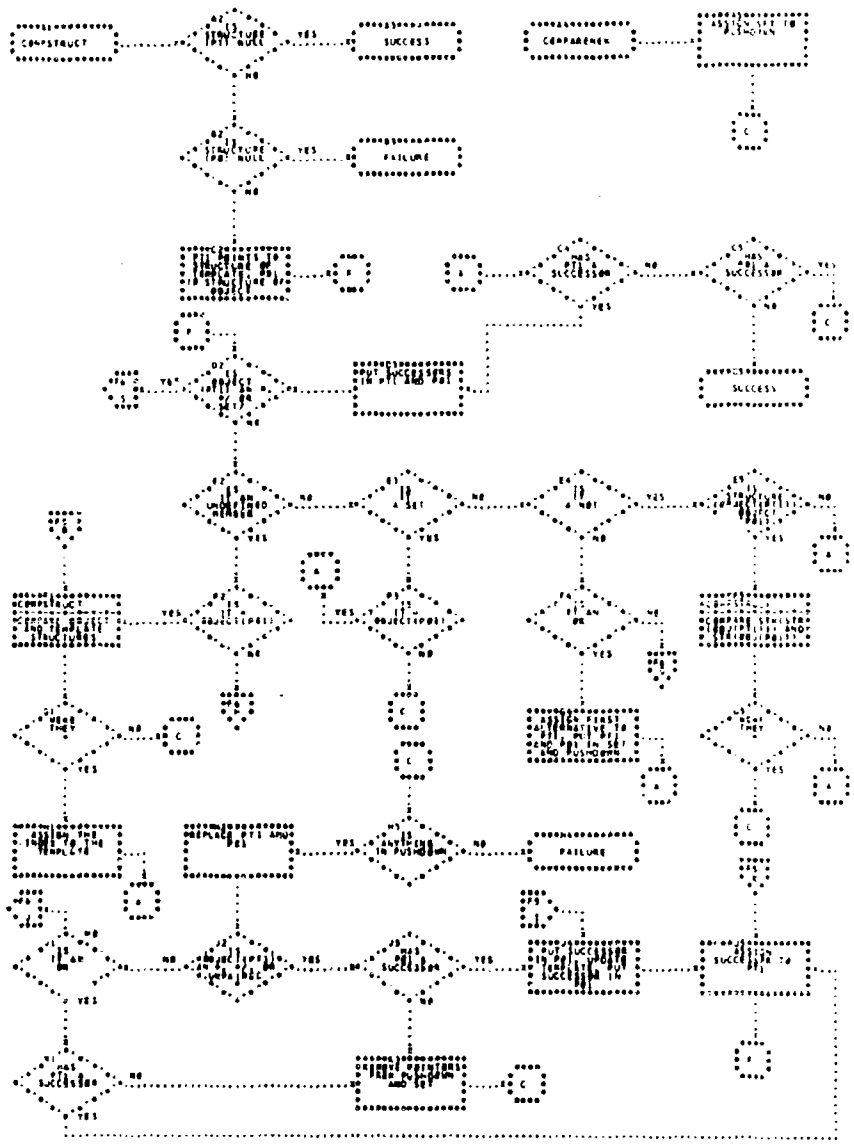
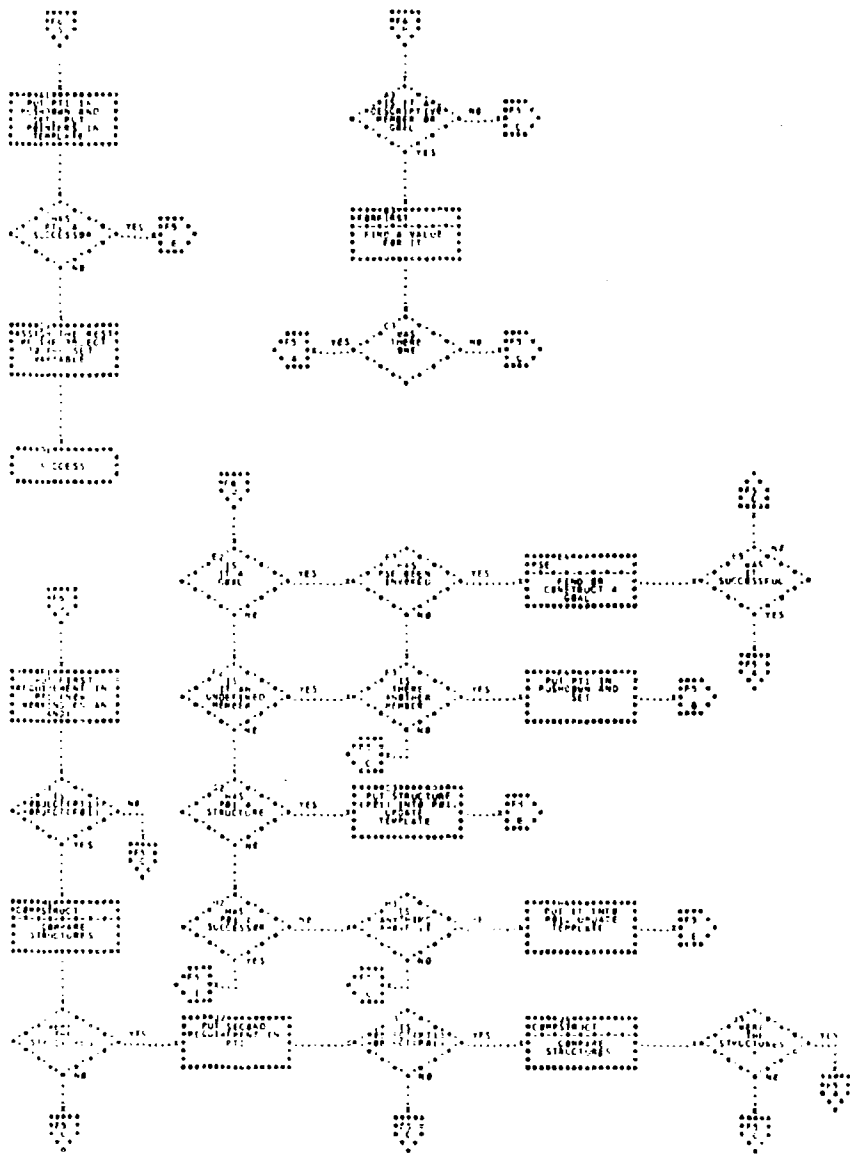


FIGURE 1. CONSTRUCT PROCEDURE





The Communicator

The Communicator contains a relatively small number of specialized procedures for communicating between the user and program - it is intended that these should be supplemented by general purpose input and output routines whose characteristics would be installation - dependent.

The procedures are designed for use with a graphic display console equipped with function buttons, keyboard, and light pen. The function buttons are used to indicate the desired operation, the keyboard to enter statements and parameters, and the light pen to indicate which statement is to be executed or which parameters are to be used in the current operation. In the following, capital italics are used to indicate the names of function buttons, and small italics to indicate statements (which may be arbitrary CL statements) and parameter lists (which are parameters separated by comma's and terminated by a semicolon).

ENTER any statement or parameter list - the function button is pushed and then any statement or parameter list may be entered from the keyboard.

ERASE any statement, parameter list, or display object - the function button is pushed and then the entity indicated by the light pen is removed from the display.

DISPLAY procedure name - the listing of the indicated procedure is displayed.

LABEL any character in a procedure listing, by any alphanumeric string enclosed in quotation marks - the button is pushed, the character is indicated by the light pen, and then the string is typed in from the keyboard. This provides a way of labeling a block for later reference, see the display and halt operations below.

DISPLAY variables or set names - the indicated variables or sets (each of which must be followed by a label if the name is not unique) are displayed. The structure and numeric and Boolean properties of a set are displayed. The members of a class may be displayed by pointing to the class name.

DISPLAYCOVER variables or set names - as above except that the cover of a set is also displayed.

HALT - halt at the current location, which is then identified.

HALT AT label - halt on entering the block with the indicated label.

CONTINUE - resume execution of the program after a halt.

EXECUTE statement - the indicated statement is executed. This is valid only if program execution has been halted.

UP, DOWN, RIGHT, LEFT, MAGNIFY, REDUCE, STOP - the display is a view from a telescope pointed at an extremely large piec of paper. These commands initiate or terminate motion of the telescope, so that, for example, one can go from page to page of a long listing. Newly displayed material displaces old material upward, so that one can return to a previous display by pushing the *UP* button.

SLOWER, FASTER - these buttons govern the rate at which the telescope is moved.

The display, displaycover, and halt procedures may also be invoked from the program. See Table 2 for more details.

System Procedures

Table 2 summarizes the characteristics of the system procedures which have been introduced in this and earlier Chapters. The procedures copy, destroy 1, and destroy 0, which are referred to in Chapter V, are also presented.

TABLE 2

SYSTEM PROCEDURES

Name	Parameter list	Explanation
create.	set	The value of the procedure is the set which is created
copy	set	The value of the procedure is the copy
destroy.	set	Destroys a set and all the sets containing it
destroy 1	set	Returns a set, and all the pointers from it or to it, to free storage

destroy 0	set	Returns a set, and all the pointers from it, to free storage
reclamation	-	Aborts the program
and	[template],[template]	See earlier section on CL for and PSE
or	[condition list], [condition list]	"
not	[template]	"
foreach	-	Value is true or false
forfirst	pointer to template	"
PSE	pointers to object and template	"
COMPSTRUCT	"	"
COMPARENEW	"	"
halt	message	Halts with message to user
display	set	Displays the current value of the set to the user. Numeric and Boolean properties and structure are displayed where appropriate
displaycover	set	As above, but cover is also displayed

CHAPTER V
IMPLEMENTATION

The Class Language has been designed to be a fairly easily implemented extension of ALGOL. The syntax may appear to be quite formidable, but could be very much simplified without effecting anything except the diagnostic capabilities of the compiler and the intelligibility of the programmer's manual. Even as presented, though, it should present no particular problems; a large amount of work in recent years has provided quite adequate tools for constructing syntax analyzers (see the excellent review by Feldman and Gries, 1968). Hence this Chapter will concentrate upon the more critical area of semantics.

Most of the new constructs in CL result in the production of data structures, code to initialize data structures, and ALGOL code to check Boolean conditions, set the values of variables, or to invoke various CL procedures. Relatively little other code is produced. This has the following advantages: the System will be easily implemented by means of a pre-processor to produce ALGOL; the System will be easily altered (such major sections as the PSE are independent procedures, as discussed in Chapter IV and below);the size of program code is limited as much as possible (this is important, since very simple CL constructs can involve extremely large amounts of processing, as shown in the flowcharts of Chapter IV). Loss of time, the usual disadvantage of systems implemented via procedure calls, is relatively unimportant, because the overhead involved is small compared with the total processing performed.

The implementation of each of the parts of the System will now be discussed. The aim is primarily to show feasibility, so the discussions will be rather brief.

Declarations have the following effects: an entry is made in a symbol table and an appropriate data structure is constructed, or a standard condition is established for later reference.

Structural expressions will result in the production of actual code sequences, since these sequences are relatively short and should be as fast as possible. A promising approach is to alter an ALGOL compiler somewhat to accept a new data

type, the set, and produce the appropriate code for structural expressions. No particular difficulty is anticipated in this, since the operations are simple (or are ALGOL - like combinations of simple operations) and have close ALGOL parallels.

The create. procedure is described below in terms of a simpler procedure called copy, which is called with an undefined member as its parameter; copy's value is a member which has a defined index, and is otherwise a copy of the parameter. Note that create. is recursive.

```
set procedure create.(set); set set;
begin set A, B;
if type (set) = undefinedmember then
begin comment the type of a set is a built-in property with the obvious
significance;
B:= copy (set);
for each A in structure (set) do
structure (B):= structure (B) + [create. (A)];
create.:= B
end
else
create.:= set
end create.
```

The destroy. procedure is implemented using a simpler procedure, destroy 0. which was described in Table 2. The destroy 1 procedure may be used instead of destroy. in certain applications where the objects *containing* the destroyed object are not to be destroyed. The destroy. and destroy 1 procedures use free storage, so it may be necessary to re-code them if the reclamation procedure is altered.

```
procedure destroy. (set); set set,
begin set A, B;
A:= cover (set);
```

```
destroy 0 (set);
comment now destroy the sets containing the set.;
for each B in A do
begin A:= A + cover (B) - [B];
    destroy 0(B) end
end destroy.;
procedure destroy 1 (set); set set;
begin set A;
comment the procedure returns set. and all the pointers from it or to it,
to free storage.;
for each A in structure (set) do
    cover (A):= cover (A) - [set];
comment now return set, and all the pointers from it, to
free storage.;
destroy 0(set)
end destroy 1,
```

Various schemes could be used to implement the fetching or storing of property values. The simplest would be to code two procedures, say fetch and store, which would have as arguments the set and the property. A more efficient approach would be to have two sets of fetch/store procedures. one to handle sets and classes of declared cardinality (which require no searching). and one to handle classes without declared cardinality (the required member must be found on a list). A still better approach would be to produce a specialized fetch/store for each set and/or property. Present plans call for either the first or second approach in the initial implementation of the System.

The PSE and one form of the CL for are discussed at length in Chapter IV. The other form of the CL for, involving the assignment of one set after another to a given set name, is comparatively simple - all that is required is a single pointer to the current position in a structure list.

Edit statements are, basically, only abbreviations of CL for statements, so

should be readily implemented. The construction of local blocks, necessary for editing and abstraction, is readily accomplished by a slight variation of the copy procedure.

Finally, the Communicator presents special problems in its implementation. Display of information should not be difficult, since the structures to be displayed are already in forms suitable for processing by display procedures. Execution of a statement entered from the graphic keyboard requires, however, the abilities to call the compiler during the running of a program and to pass a symbol table from one invocation of the compiler to another. The difficulties involved seem great, but not insurmountable; the advantages of such an interactive system are sufficient to far offset them. Further, although the rest of the System is designed to permit graceful improvement, the interactive features will have to be built in from the beginning if the System is not to be completely redone to add them.

The foregoing demonstrates, hopefully, that the implementation of the System, while not a trivial task, is at least reasonably well-defined and feasible.

CHAPTER VI
CONCLUSION

This Chapter begins with a discussion of various problems for which the Heuristic Programming System might be used. Experiments with the Go-Moku program are suggested, related problem areas are discussed, and then a rather unrelated area is discussed. Next, a problem area for which the System is unsuitable is discussed. The Chapter concludes with a discussion of features which might be added to the System.

The Go-Moku program has various parameters which undoubtedly greatly effects its performance. Playing time, the distribution of playing time between move generation and evaluation, the initial subgoal descriptions, the initial board configuration, and the strategy used against it, may all be varied by an experimenter. The extent to which the program's performance is sensitive to small variations in these parameters would be of great interest, especially in the design of similar programs. Further experimentation could be carried out by altering the generatemove and minimax procedures. The way in which the available search time is allocated in generatemove might, for example, be made to depend upon the total amount of search time available, the number of goals, and so on. The procedure might also save time by checking first for goals which might reasonably be present - e.g., those with values close to the evaluation of the previous move - and then spending any extra time looking for unlikely configurations of high value. Also, it may soon become necessary, as more and more descriptions are created, to attempt to generalize so as to reduce their number, or to destroy those which are less valuable. The minimax procedure might be improved considerably, since at present, for simplicity, it uses generatemove and one class of goals to generate feasible moves and to evaluate the terminal positions - better play might result from two procedures and/or two classes of goals. More complex criteria might also be used to determine terminal positions - e.g., the evaluate procedure might be extended to look for more forcing positions. An improvement with potentially great effects

would be the re-writing of the section of code which generates descriptions of new subgoals. It would be especially desirable to be able to incorporate the relevant pieces (stones) of *both* sides into the descriptions, so that the depth of minimax searching necessary for a given level of play could be substantially decreased. The problem, of course, is in deciding which pieces, and which unoccupied points, are really relevant. However, this is primarily a problem in analysis and experimentation, not in programming - experience with the System has so far shown that, once a process is thoroughly understood, it can be readily programmed.

As a rather important aside, note the very considerable modularity of the Go-Moku program - alterations could be made independently in the procedures mentioned above and in the improvement code. Furthermore, the minimax and generatemove procedures are rather independent of Go-Moku as well! This is of tremendous importance to the user of the System, since he can readily divide a complex problem like Go-Moku into a number of simpler parts, whose interactions are readily apparent to him. This is not an accidental property of this particular program, but a part of the very nature of the Heuristic Programming System - the hierarchical organization of objects leads rather easily to the hierarchical organization of the programs which manipulate them.

Go-Moku appears to be a rather promising vehicle for investigating human problem-solving abilities - the game is easy to learn yet the play can become quite complex, and the possibilities for pattern recognition and description are numerous and apparent to even novice players. Particularily in the fields of learning and pattern recognition, Go-Moku seems considerably richer than any of the problem domains explored by GPS (Newell, Shaw, Simon, 1960a, and Ernst and Newell, 1967). The preceding comments have presented the parameters and procedures which might be modified in attempting to make the program either more or less human. A particularly interesting series of experiments might be conducted to try to improve the similarity to human play at various levels of experience by adjusting the number of subgoals, the

amount of search time, and the depth of minimax. The results might be indicative of the relative importance of pattern recognition and planning, on the one hand, and mental experimentation, on the other, in human play.

The Go-Moku program has been discussed in great detail above because it illustrates the use of the Heuristic Programming System in programming some of the tools necessary for artificial intelligence: search, pattern recognition, and learning. Each of these exists as a problem area in its own right, so the application of the System to each will now be discussed.

The search technique used in the program was a very specialized type of hill-climbing, implemented by means of the generation of feasible moves and the minimax procedure. As indicated in Chapter I, other search techniques might also be used: all that is required, basically, is a means for choosing among various possibilities. and then keeping records of the effects of each trial. Various mathematical techniques for finding local minima or maxima - such as the method of steepest descent or the simplex method - may be easily used if applicable, since ALGOL is a subset of the System. Record keeping is greatly facilitated by the non-ALGOL features of the System. Note that a pure list system such as LISP 1.5 (McCarthy *et al.*, 1965) would be inefficient in problems requiring large amounts of arithmetic, while a mathematical language would be at least inconvenient, and possibly very inefficient, at record-keeping. Note further the important interactions between pattern recognition and search - the former may be used, for example, to choose a technique to be used by the latter.

Pattern recognition, as an independent area of interest, also profits from the mixture of numeric and non-numeric processing made convenient by the System. For example, one might construct a photo-interpretation scheme in which a two-dimensional correlation technique would be used to locate objects of possible interest, these objects would be further classified according to complex relationships among them, and, finally, combined hierarchically to form a general interpretation of the entire scene. The block structures of the System

provide a convenient and efficient means for storing all of the various data necessary for such a scheme. Note also the possibility of constructing the scheme in such a way that if ambiguous or nonsensical results were generated at any level, then a previous level could be returned to for reprocessing. This sort of thing is difficult to foresee when first constructing a program so is usually added later, if at all. The interactive features of the System should aid the user in recognizing when program modification should occur, and the modularity of programming should aid in implementing the modification. Note that the ability to declare initial property values and standard tests contributes greatly to the ease with which afterthoughts may be added to a program. The flexibility of pattern matching provided by the PSE facilitates both the original programming and the modifications.

The third area, learning, may be very close to pattern recognition, as in the generation of new descriptions in the Go-Moku program. Other forms of learning might also be implemented using the System - in fact, assigning minimax values to moves to increase the number of cutoffs demonstrates short-term learning of a rather useful sort. However, the major emphasis of the System is toward pattern description and recognition, which rather obviously lie at the heart of any advanced learning system. Also rather obviously, the Go-Moku program represents only a very small application of a general process: recognition of an interesting situation, isolation of the relevant objects, and formation of a description of them. As noted in Chapter I, the ease with which this process may be carried out is very dependent upon the representation being used. In the Go-Moku program, for example, the process was rather simple because the relevant objects were easily isolated and described; more complex programs, possibly involving a great amount of interaction with the human user, will be required in other games, such as checkers or chess. While the amount of effort to be expended is apt to be quite large, the results should be much superior to those obtainable by parameter-adjustment programs such as Samuel's checker player (1959).

The Traveling Salesman problem and various resource allocation problems in integer programming are examples of problems rather unrelated to Go-Moku but suitable for solution using the System. These problems typically involve large amounts of data, various trial and error procedures which are efficient in some cases and inefficient in others, and a rather intimate mix of numeric and non-numeric data processing. Interaction with the user may also be important. The System could be very profitably used for such problems: the pattern recognition features might be used to choose among possible solution methods, the block structure of the data to record various solution attempts, and the ALGOL base to do the numeric processing.

In contrast to preceding problem areas, the standard problems of symbol manipulation (theorem proving and symbolic integration, for example) seem ill-suited to the System: the difficulty is that the System has no built-in relation representing concatenation. Thus, the fact that one symbol is to the right of another must be represented by a special class member containing both symbols. This is inefficient in both time and storage, and extremely inconvenient to the programmer. Input and output would doubtless also be inconvenient and inefficient. Symbol manipulation might be attempted if part of a more suitable problem, but should definitely be discouraged in general.

Possible additions and modifications to the System include the refinement of the CL for and the PSE to make their searches more efficient; a more flexible mechanism in the PSE to choose the descriptive member to be used in constructing a goal; a more general and convenient improvement mechanism; the implementation of some automatic improvement features (such as keeping a record of the results of using each descriptive member); and additional macro facilities to ease the burden of propagating changes from one object to another. More flexibility in the structure of data blocks would be desirable. For example, structure and cover might be optional, or new types of pointers might be used to link objects together, or to order objects in various ways.

Other changes might be considered, but should not detract from the

present convenience, flexibility, and efficiency of the System.

BIBLIOGRAPHY

The **Communications of the Association for Computing Machinery** is abbreviated as CACM. The **Journal of the Association for Computing Machinery** is abbreviated as **JACM**.

Amarel, S., 1962. "On the automatic formation of a computer program that represents a theory." **Self-Organizing Systems - 1962**. Edited by M. Yovits, G. Jacobi, and G. Goldstein. Washington:Spartan Books.

Amarel, S., 1968. "On representations of problems of reasoning about actions." **Machine Intelligence 3**. Edited by D. Michie. New York:American Elsevier Publishing Company, Inc.

Elcock, E. and Murray, A., 1967. "Experiments with a learning component in a Go-Moku playing program." **Machine Intelligence 1**. Edited by N. Collins and D. Michie. New York:American Elsevier Publishing Company.

Ernst, G. and Newell, A., 1967. "Some issues of representation in a general problem solver." **Proc. Spring Joint Computer Conference30**:583-600.

Farber, D., Griswold, R., and Polonsky, J., 1964. "SNOBOL, A string manipulation language," **JACM**, **11**(2):21-30.

Feldman, J. A. and Gries, D., 1968. "Translator writing systems,"**CACM**, **11**(2):77-113.

Forte, A., 1967. **SNOBOL 3 Primer**. Cambridge:The MIT Press.

Holland, J., 1962, "Outline for a logical theory of adaptive systems," **JACM**, **9**(3):297-314.

Hormann, A., 1962. "Programs for machine learning. Part I," **Information and Control**, 5(4):347-367.

Hormann, A., 1962. "Programs for machine learning. Part II," **Information and Control**, 7(1):55-77.

Hormann, A., 1965. "Gaku: an artificial student," **Behavioral Science**, 10:88-107.

Koffman, E., 1967. "Learning through pattern recognition applied to a class of games," Case-Western Reserve University, Cleveland, Ohio, Systems Research Center Report SRC 107-A-67-45, 1967.

Koffman, E., 1968. "Learning games through pattern recognition," **IEEE Transactions on Systems Science and Cybernetics**, SSC-4(1):12-16.

Lasker, E., 1960. **Go and Go-Moku**. New York:Dover Publications, Inc.

McCarthy, J., *et al.*, 1965. **LISP 1.5 Programmer's Manual**. Cambridge:The MIT Press.

Minsky, M., 1961. "Descriptive languages and problem solving," **Proc. Western Joint Computer Conference**, 19:215-218.

Minsky, M., 1963. "Steps Toward Artificial Intelligence." **Computers and Thought**. Edited by E. Feigenbaum and J. Feldman. New York:McGraw-Hill Book Company.

Murray, A. and Elcock, E., 1968, "Automatic description and recognition of board patterns in Go-Moku." **Machine Intelligence 2**. Edited by E. Dale and D. Michie. New York:American Elsevier Publishing Company, Inc.

Naur, P., *et al.*, 1963. "Revised report on the algorithmic language ALGOL 60," **CACM**, 6(1):1-17.

Newell, A., Shaw, J., and Simon, H., 1957. "Empirical explorations of the logic theory machine," **Proc. Western Joint Computer Conference**, 11:218-239. (Also in **Computers and Thought**. Edited by E. Feigenbaum and J. Feldman. New York:McGraw-Hill Book Company, 1963.)

Newell, A., Shaw, J., and Simon, H., 1960a. "Report on a general problem solving program." **Information Processing: Procedings of the International Conference on Information Processing, UNESCO, Paris, 15-20 June 1959**. Paris:UNESCO.

Newell, A., Shaw, J., and Simon, H., 1960b. "A variety of intelligent learning in a general problem solver." **Self-Organizing Systems**. Edited by M. Yovits and S. Cameron. New York:Pergamon Press.

Reynolds, J., 1965. "An Introduction to the COGENT programming system," **Proc 20th National Conference, Association for Computing Machinery**:422-436.

Roberts, L., 1965. "Graphical communication and control languages." **Second Congress on the Information System Sciences**. Washington:Spartan Books.

Ross, D., 1967a. "The AED approach to generalized computer-aided design," **Proc. 22nd National Conference, Association for Computing Machinery**:367-385.

Ross, D., 1967b. "The AED free storage package," **CACM** 10(8):481-491.

Ross, D., 1968. **AED-0 Programmer's Guide**. Computer-Aided Design Project, Electronic Systems Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.

Rovner, P., and Feldman, J. A., 1967. "An associative processing system for conventional digital computers," Technical Note 1967-19, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, Mass.

Samuel, A., 1959, "Some studies in machine learning, using the game of checkers," **IBM Journal of Research and Development**, 3(3):210-229. (Also in **Computers and Thought**. Edited by E. Feigenbaum and J. Feldman. New York:McGraw-Hill Book Company, 1963.)

Slagle, J., and Bursky, P., 1968. "Experiments with a multipurpose, theorem-proving heuristic program." **JACM**, 15(1):85-99.

Sutherland, I., 1963. "Sketchpad: a man machine graphical communication system," **Proc. Spring Joint Computer Conference**, 23:329-346.

APPENDIX

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) U. S. Naval Weapons Laboratory Dahlgren, Virginia	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

A HEURISTIC PROGRAMMING SYSTEM

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

5. AUTHOR(S) (First name, middle initial, last name)

David K. Jefferson

6. REPORT DATE April 1969	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) TR-2281 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	

10. DISTRIBUTION STATEMENT

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

The Heuristic Programming System is a tool for research in many areas of artificial intelligence, particularly pattern recognition and adaptive systems. It provides the arithmetic capabilities and recursive structure of ALGOL plus flexible and efficient facilities for representing and manipulating complex hierarchically structured objects. Objects may be created, modified, destroyed, or described by other, descriptive, objects. A search operation can retrieve objects or collections of objects which are specified by arbitrarily complex descriptions. Another search operation can not only retrieve objects, but can construct them according to the specifications of previously created descriptive objects; this greatly facilitates the implementation of self-improving pattern recognition schemes, which are basic to advanced work in artificial intelligence.

The report contains a discussion of the programming facilities required for artificial intelligence, an informal introduction to the System, a formal programmer's manual with numerous examples, a sample program which plays the game of Go-Moku, and a discussion of a proposed implementation.

DD FORM 1 NOV 65 1473 (PAGE 1)

UNCLASSIFIED

Security Classification